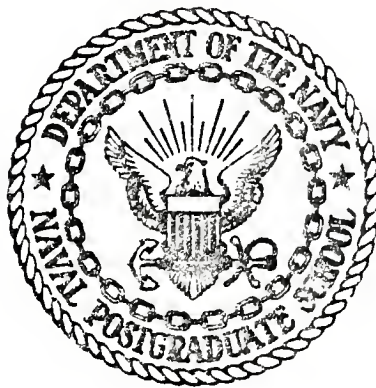


MICRO-COBOL
A SUBSET OF
NAVY STANDARD HYPO-COBOL
FOR MICRO-COMPUTERS

Philip Russell Mylet

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

MICRO-COBOL
A SUBSET OF
NAVY STANDARD HYPO-COBOL
FOR MICRO-COMPUTERS

by

Philip Russell Mylet

September 1978

Thesis Advisor:

G. A. Kildall

Approved for public release; distribution unlimited.

T18-073

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MICRO-COBOL a Subset of Navy Standard Hypo-Cobol for Micro-Computers		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; September 1978
7. AUTHOR(s) Philip Russell Mylet		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE September 1978
		13. NUMBER OF PAGES 169
		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) MICRO-COBOL Navy Standard Hypo-Cobol Micro-Computers Compiler		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A MICRO-COBOL interpretive compiler has been implemented on an 8080 micro-computer based system running under CP/M. The implementation is a subset of ADPESO standard HYPO-COBOL in that the interprogram communication module has not been included. HYPO-COBOL provides nucleus level constructs and file options from the ANSI COBOL package along with the		

DD FORM 1473
1 JAN 73EDITION OF 1 NOV 68 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

PERFORM UNTIL construct from a higher level to give increased structural control. MICRO-COBOL can be executed on an 8080 or Z-80 micro-computer system with 16K of memory. Although largely completed and tested, all features are not implemented. File I/O features have not been tested and the numeric edit instruction has not been implemented in the interpreter.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Approved for public release; distribution unlimited.

MICRO-COBOL
A Subset of
Navy Standard HYPO-COBOL
for Micro-Computers

by

Philip Russell Mylet
B.S., Pennsylvania State University, 1967

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1978

ABSTRACT

A MICRO-COBOL interpretive compiler has been implemented on an 8080 micro-computer based system running under CP/M. The implementation is a subset of ADPESO standard HYPO-COBOL in that the interprogram communication module has not been included. HYPO-COBOL provides nucleus level constructs and file options from the ANSI COBOL package along with the PERFORM UNTIL construct from a higher level to give increased structural control. MICRO-COBOL can be executed on an 8080 or Z-80 micro-computer system with 16K of memory. Although largely completed and tested, all features are not implemented. File I/O features have not been tested and the numeric edit instruction has not been implemented in the interpreter.

TABLE OF CONTENTS

I.	INTRODUCTION -----	7
	A. BACKGROUND -----	7
	B. APPROACH -----	7
II.	MICRO-COBOL INTERPRETER -----	10
	A. GENERAL DESCRIPTION -----	10
	B. MEMORY ORGANIZATION -----	11
	C. INTERPRETER INSTRUCTIONS -----	11
	1. Format -----	11
	2. Arithmetic Operations -----	12
	3. Branching -----	13
	4. Moves -----	16
	5. Input-output -----	19
	6. Special Instructions -----	22
III.	MICRO-COBOL COMPILER -----	25
	A. GENERAL -----	25
	B. CONTROL FLOW -----	25
	C. INTERNAL STRUCTURES -----	25
	D. PART ONE -----	27
	E. PART TWO -----	35
	APPENDIX A -----	45
	APPENDIX B -----	93
	APPENDIX C -----	97
	APPENDIX D -----	99
	COMPUTER LISTINGS -----	101
	LIST OR REFERENCES -----	168
	INITIAL DISTRIBUTION LIST -----	169

ACKNOWLEDGMENTS

I wish to express my appreciation to my advisor, Gary Kildall who cheerfully accepted the responsibilities of thesis advisor while on leave of absence. My thanks also to John Pierce of Digital Research for his contributions and hours of assistance early in the project. Finally, I wish to express my gratitude to Mark Moranville who continuously provided technical assistance and moral support during the times when it was most needed.

I. INTRODUCTION

A. BACKGROUND

MICRO-COBOL is an implementation of ADPESO standard MYPO-COBOL with the major exception that the interprogram communication module is not included. It has been implemented as an interpretive compiler in that the compiler itself generates intermediate code which is then executed by a separate interpreter program. Both compiler and interpreter run under CP/M on an 8080 or Z-80 micro-computer system with 16K of memory. Much credit for this work goes to Allen S. Craig who did the original design and implementation of MICRO-COBOL for his thesis submitted in March 1977. Craig's work is contained in Reference 1. Most of the coding had been completed, but many of the constructs did not work or worked incorrectly. Since much of the compiler had not been debugged and some areas not completed, thesis work was continued in March 1978 with the goal of producing a working MICRO-COBOL compiler and interpreter.

B. APPROACH

As a first step, the program listings and thesis were studied to gain familiarity with the original project goals and resolve several areas of conflict between the thesis and the listings. The remaining effort consisted of running test programs, isolating bugs, and making additions, corrections

and small design changes. The problems discovered were primarily errors in the code, however, there were also missing routines and grammar problems which necessitated reconstructing the original grammar. Appendix D lists the features that did not work at the start of this project and the bugs that are known to remain.

The HYPO-COBOL Compiler Validation System (HCCVS) was obtained from the Automatic Data Processsing Equipment Selection Office (ADPESO) to be used in testing the compiler. The HCCVS is intended to determine the degree to which the individual language elements conform to the HYPO-COBOL Specification. The validation system is made up of audit routines, their related data, and an executive routine which prepares the audit routines for compilation. Each audit routine is a HYPO-COBOL program which includes tests and supporting procedures that print out the results of each test. The audit routines collectively test the features of the HYPO-COBOL Language Specification. Since MICRO-COBOL does not support the interprogram communication module feature of HYPO-COBOL, the HCCVS is not useful in its existing form; however, it contains numerous routines which can be used to create small test programs that should run on MICRO-COBOL as it currently exists.

A language construct in question was tested by writing a test program, compiling it, and executing it on the interpreter. If problems were encountered, the intermediate code

was examined to determine if the difficulty was in the compiler or the interpreter. Having made this determination, the program was examined to isolate the bad code using SID (see Reference 12). Changes were then made and the source program recompiled using the ISIS editor and the PLM80 compiler on the INTEL MDS System. Appendix B describes the procedure used to construct the executable compiler and interpreter files from the edited PLM80 source files.

The following sections describe the implementation of the compiler and interpreter. This material should be read in conjunction with Reference 1 which contains additional background information.

II. MICRO-COBOL INTERPRETER

A. GENERAL DESCRIPTION

The following sections describe the MICRO-COBOL pseudo-machine architecture in terms of allocated memory areas and pseudo-machine operations. The machine operators contain all of the information required to perform one complete action required by the language. The machine contains multiple parameter operators and a program counter that addresses the next instruction to be executed. Three eighteen digit registers are used for arithmetic and logic operations. A subscript stack is used to compute subscript locations, and a set of flags are used to pass branching information from one instruction to another. The registers allow manipulation of signed numbers of up to eighteen decimal digits in length. Included in their representation is a sign indicator and the position of the assumed decimal point for the currently loaded number. The HYPO-COBOL specification requires that there be no loss of precision for operations on numbers having eighteen significant digits. Numbers are represented in "DISPLAY" and "packed decimal" formats. DISPLAY format numbers are represented in memory in ASCII and may have separate signs indicated by "+" and "-" or may have a "zone" indicator, denoting a negative sign. In packed decimal format the numbers are represented in memory as sequential digit pairs and the sign is indicated in the right-most position.

B. MEMORY ORGANIZATION

Memory is divided into three major sections: (1) the data areas defined by the DATA DIVISION statements, (2) the code area, (3) and the constants area. No particular order of these sections is required. The first two areas assume the ability to both read and write, but the third only requires the ability to be read. The code area requires write capability because several instructions store branch addresses and return addresses during execution.

The data area contains variables defined by the DATA DIVISION statements, constants set in the WORKING STORAGE SECTION, and all file control blocks and buffers. These elements will be manipulated by the machine as each instruction is executed.

C. INTERPRETER INSTRUCTIONS

1. Format

All of the interpreter instructions consist of an instruction number followed by a list of parameters. The following sections describe the instructions, list the required parameters, and describe the actions taken by the machine in executing each instruction. In each case, parameters are denoted informally by the parameter name enclosed in brackets. The BRN branching instruction, for example, uses the single parameter <branch address> which is the target of the unconditional branch.

As each instruction number is fetched from memory, the program counter is incremented by one. The program counter is then either incremented to the next instruction number, or a branch is taken.

The three eighteen digit registers which are used by the instructions covered in the following section are referred to as registers zero, one, and two.

2. Arithmetic Operations

There are five arithmetic instructions which act upon the three registers. In all cases, the result is placed in register two. Operations are allowed to destroy the input values during the process of creating a result, therefore, a number loaded into a register is not available for a subsequent operation.

ADD: (addition). Sum the contents of register zero and register one.

Parameters: no parameters are required.

SUB: (subtract). Subtract register zero from register one.

Parameters: no parameters are required.

MUL: (multiply). Multiply register zero by register one.

Parameters: no parameters required.

DIV: (divide). Divide register one by the value in register zero. The remainder is not retained.

Parameters: no parameters are required.

RND: (round). Round register two to the last significant significant decimal place.

Parameters: no parameters are required.

3. Branching

The machine contains the following flags which are used by the conditional instructions covered in this section.

BRANCH flag -- indicates if a branch is to be taken;

END OF RECORD flag -- indicates that an end of input condition has been reached when an attempt was made to read input;

OVERFLOW flag -- indicates the loss of information from a register due to a number exceeding the available size;

INVALID flag -- indicates an invalid action in writing to a direct access storage device.

All of the branch instructions are executed by changing the value of the program counter. Some are unconditional branches and some test for condition flags which are set by other instructions. A conditional branch is executed by testing the branch flag which is initialized to false.

A true value causes a branch by changing the program counter to the value of the branch address. The branch flag is then reset to false. A false value causes the program counter to be incremented to the next sequential instruction.

BRN: (branch to an address). Load the program counter with the <branch address>.

Parameters: <branch address>

The next three instructions share a common format.

The memory field addressed by the <memory address> is checked for the <address length>, and if all the characters match the test condition, the branch flag is complemented.

Parameters: <memory address> <address length> <branch address>

CAL: (compare alphabetic). Compare a memory field for alphabetic characters.

CNS: (compare numeric signed). Compare a field for numeric characters allowing for a sign character.

CNU: (compare numeric unsigned). Compare a field for numeric characters only.

DEC: (decrement a counter and branch if zero).

Decrement the value of the <address counter> by one; if the result is zero before or after the decrement, the program counter is set to the <branch address>. If the result is not zero, the program counter is incremented by four.

Parameters: <address counter> <branch address>

EOR: (branch on end-of-records flag). If the end-of-records flag is true, it is set to false and the program counter is set to the <branch address>. If false, the program counter is incremented by two.

Parameters: <branch address>

GDP: (go to - depending on). The memory location addressed by the <number address> is read for the number of bytes indicated by the <memory length> . This number indicates which of the <branch addresses> is to be used.

The first parameter is a bound on the number of branch addresses. If the number is within the range, the program counter is set to the indicated address. An out-of-bounds value causes the program counter to be advanced to the next sequential instruction.

Parameters: <bound number - byte> <memory length> <memory address> <branch addr-1> <branch addr-2> ... <branch addr-n>

INV: (branch if invalid-file-action flag true). If the invalid-file-action flag is true, then it is set to false, and the program counter is set to the branch address. If it is false, the program counter is incremented by two.

Parameters: <branch address>

PER: (perform). The code address addressed by the <change address> is loaded with the value of the <return address>. The program counter is then set to the <branch address>.

Parameters: <branch address> <change address> <return address>

RET: (return). If the value of the <branch address> is not zero, then the program counter is set to its value, and the <branch address> is set to zero. If the <branch address> is zero, the program counter is incremented by two.

Parameters: <branch address>

REQ: (register equal). This instruction checks for a zero value in register two. If it is zero, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RGT: (register greater than). Register two is checked for a negative sign. If present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

SER: (branch on size error). If the overflow flag is true, then the program counter is set to the branch address, and the overflow flag is set to false. If it is false, then the program counter is incremented by two.

Parameters: <branch address>.

The next three instructions are of similar form in that they compare two strings and set the branch flag if the condition is true.

Parameters: <string addr-1> <string addr-2> <length - address> <branch address>

SEQ: (strings equal). The condition is true if the strings are equal.

SGT: (string greater than). The condition is true if string one is greater than string two.

SLT: (string less than). The condition is true if string one is less than string two.

4. Moves

The machine supports a variety of move operations for various formats and types of data. It does not support direct moves of numeric data from one memory field to another. Instead, all of the numeric moves go through the registers.

The next seven instructions all perform the same function. They load a register with a numeric value and

differ only in the type of number that they expect to see in memory at the <number address>. All seven instructions cause the program counter to be incremented by five. Their common format is given below.

Parameters: <number address> <byte length> <byte decimal count> <byte register to load>

LOD: (load literal). Register two is loaded with a constant value. The decimal point indicator is not set in this instruction. The literal will have an actual decimal point in the string if required.

LD1: (load numeric). Load a numeric field.

LD2: (load postfix numeric). Load a numeric field with an internal trailing sign.

LD3: (load prefix numeric). Load a numeric field with an internal leading sign.

LD4: (load separated postfix numeric). Load a numeric field with a separate leading sign.

LD5: (load separated prefix numeric). Load a numeric field with a separate trailing sign.

LD6: (load packed numeric). Load a packed numeric field.

MED: (move into alphanumeric edited field). The edit mask is loaded into the <to address> to set up the move, and then the <from address> information is loaded. The program counter is incremented by ten.

Parameters: <to address> <from address> <length of move>
<edit mask address> <edit mask length>

MNE: (move into a numeric edited field). First the edit mask is loaded into the receiving field, and then the information is loaded. Any decimal point alignment required will be performed. If truncation of significant digits is a side effect, the overflow flag is not set. The program counter is incremented by twelve.

Parameters: <to address> <from address> <address length of move> <edit mask address> <address mask length> <byte to decimal count> <byte from decimal count>

MOV: (move into an alphanumeric field). The memory field given by the <to address> is filled by the from field for the <move length> and then filled with blanks in the following positions for the <fill count>.

Parameters: <to address> <from address> <address move length> <address fill count>

STI: (store immediate register two). The contents of register two are stored into register zero and the decimal count and sign are indicators set.

Parameters: none.

The store instructions are grouped in the same order as the load instructions. Register two is stored into memory at the indicated location. Alignment is performed and any truncation of leading digits causes the overflow flag to be set. All five of the store instructions cause the program counter to be incremented by four. The format for these instructions is as follows.

Parameters: <address to store into> <byte length> <byte decimal count>

MNE: (move into a numeric edited field). First the edit mask is loaded into the receiving field, and then the information is loaded. Any decimal point alignment required will be performed. If truncation of significant digits is a side effect, the overflow flag is not set. The program counter is incremented by twelve.

Parameters: <to address> <from address> <address length of move> <edit mask address> <address mask length> <byte to decimal count> <byte from decimal count>

MOV: (move into an alphanumeric field). The memory field given by the <to address> is filled by the from field for the <move length> and then filled with blanks in the following positions for the <fill count>.

STI: (store immediate register two). The contents of register two are stored into register zero and the decimal count and sign are indicators set.

Parameters: none.

The store instructions are grouped in the same order as the load instructions. Register two is stored into memory at the indicated location. Alignment is performed and any truncation of leading digits causes the overflow flag to be set. All five of the store instructions cause the program counter to be incremented by four. The format for these instructions is as follows.

Parameters: <address to store into> <byte length> <byte decimal count>

ST0: (store numeric). Store into a numeric field.

ST1: (store postfix numeric). Store into a numeric field with an internal trailing sign.

ST2: (store prefix numeric). Store into a numeric field with an internal leading sign.

ST3: (store separated postfix numeric). Store into a numeric field with a separate trailing sign.

ST4: (store separated prefix numeric). Store into a numeric field with a separate leading sign.

ST5: (store packed numeric). Store into a packed numeric field.

5. Input-Output

The following instructions perform input and output operations. Files are defined as having the following characteristics: they are either sequential or random and, in general, files created in one mode are not required to be readable in the other mode. Standard files consist of fixed length records, and variable length files need not be readable in a random mode. Further, there must be some character or character string that delimits a variable length record.

ACC: (accept). Read from the system input device into memory at the location given by the memory address . The program counter is incremented by three.

Parameters: <memory address> <byte length of read>

CLS: (close). Close the file whose file control block is addressed by the <fcb address>. The program counter is incremented by two.

Parameters: <fcb address>

DIS: (display). Print the contents of the data field pointed to by <memory address> on the system output device for the indicated length. The program counter is incremented by three.

Parameters: <memory address> <byte length>

There are three open instructions with the same format. In each case, the file defined by the file control block referenced will be opened by the mode indicated. The program counter is incremented by two.

OPN: (open a file for input).

OP1: (open a file for output).

OP2: (open a file for both input and output). This is only valid for files on a random access device.

The following file actions all share the same format. Each performs a file action on the file referenced by the file control block. The record to be acted upon is given by the record address. The program counter is incremented by six.

Parameters: <fcb address> <record address> <record length - address>

DLS: (delete a record from a sequential file). Remove the record that was just read from the file. The file is required to be open in the input-output mode.

RDF: (read a sequential file). Read the next record into the memory area.

WTF: (write a record to a sequential file). Append a new record to the file.

RVL: (read a variable length record).

WVL: (write a variable length record).

RWS: (rewrite sequential). The rewrite operation writes a record from memory to the file, overlaying the last record that was read from the device. The file must be open in the input-output mode.

The following file actions require random files rather than sequential files. They all make use of a random file pointer which consists of a <relative address> and a <relative length>. The memory field holds the number to be used in disk operations or contains the relative record number of the last disk action. The relative record number is an index into the file which addresses the record being accessed. After the file action, the program counter is incremented by nine.

Parameters: <fcb address> <record address> <record length - address> <relative address> <relative length - byte>.

DLR: (delete a random record). Delete the record addressed by the relative record number.

RRR: (read random relative). Read a random record relative to the record number.

RRS: (read random sequential). Read the next sequential record from a random file. The relative record number of the record read is loaded into the memory reference.

RWR: (rewrite a random record).

WRR: (write random relative). Write a record into the area indicated by the memory reference.

WRS: (write random sequential). Write the next sequential record to a random file. The relative record number is returned.

6. Special Instructions

The remaining instructions perform special functions required by the machine that do not relate to any of the previous groups.

NEG: (negate). Complement the value of the branch flag.

Parameters: no parameters are required.

LDI: (load a code address direct). Load the code address located five bytes after the LDI instruction with the contents of <memory address> after it has been converted to hexadecimal.

Parameters: <memory address> <length - byte>

SCR: (calculate a subscript). Load the subscript stack with the value indicated from memory. The address loaded into the stack is the <initial address> plus an offset. Multiplying the <field length> by the number in the <memory reference> gives the offset value.

Parameters: <initial address> <field length> <memory reference> <memory length> <stack level>

STD: (stop display). Display the indicated information and then terminate the actions of the machine.

Parameters: <memory address> <length - byte>

STP: (stop). Terminate the actions of the machine.

Parameters: no parameters are required.

The following instructions are used in setting up the machine environment and cannot be used in the normal execution of the machine.

BST: (backstuff). Resolve a reference to a label.

Labels may be referenced prior to their definition, requiring a chain of resolution addresses to be maintained in the code. The latest location to be resolved is maintained in the symbol table and a pointer at that location indicates the next previous location to be resolved. A zero pointer indicates no prior occurrences of the label. The code address referenced by <change address> is examined and if it contains zero, it is loaded with the new address . If it is not zero, then the contents are saved, and the process is repeated with the saved value as the change address after loading the <new address>.

Parameters: <change address> <new address>

INT: (initialize memory). Load memory with the <input string> for the given length at the <memory address>.

Parameters: <memory address> <address length> <input string>

SCD: (start code). Set the initial value of the program counter.

Parameters: <start address>

TER: (terminate). Terminate the initialization process
and start executing code.

Parameters: no parameters are required.

III. MICRO-COBOL COMPILER

A. GENERAL

The compiler is designed to read the source language statements from a diskette, extract the needed information for the symbol table, and write the output code back onto the diskette all in one pass. The compiler is defined in two parts which run in succession. Part one builds the symbol table and leaves it in memory to be used by part two. The output from part two of the compiler is the intermediate code file.

B. CONTROL FLOW

After part one of the compiler has completed its task it loads part two without operator intervention. Internal control of the compiler is the same for both part one and two. The parser is called after initialization and runs until it either finishes its task or reaches an unrecoverable error state. The major subroutines in the compiler are the scanner and the production case statement which are both controlled by the parser.

C. INTERNAL STRUCTURES

The major internal structure is the symbol table, which was designed as a list where the elements in the list are the descriptions of the various symbols in the program. As

new symbols are encountered they are added to the end of the list. Symbols already in the list can be accessed through the use of a "current symbol pointer". The location of items in the list is determined by checking the identifier against a hash table that points to the first entry in the symbol table with that hash code. A chain of collision addresses is maintained in the symbol table which links entries which have the same hash value. All of the items in the symbol table contain the following information: a collision field, a type field, the length of the identifier, and the address of the item. If an item in the symbol table is a data field, the following information is included in the table: the length of the item, the level of the data field, an optional decimal count, an optional multiple occurrence count, and the address of the edit field, if required. If the item is a file name then the following additional information is included: the file record length, the file control block address, and the optional symbol table location of the relative record pointer. If the item is a label, then the only additional information is the location of the return instruction at the end of the paragraph or section.

In addition to the symbol table, two stacks are used for storing information: the level stack and the identifier stack. In both cases, they are used to hold pointers to entries in the symbol table. The identifier stack keeps track of multiple identifier occurrences in such statements

as the GO TO DEPENDING statement. The level stack is used to hold information about the levels that make up a record description.

The parser has control of a set of stacks that are used in the manipulation of the parse states. In addition to the state stack that is required by the parser, part one has a value stack while part two has two different value stacks that operate in parallel with the parser state stack. The use of these stacks is described below.

D. PART ONE

The first part of the compiler is primarily concerned with building the symbol table that will be used by the second part. The actions corresponding to each parse step are explained in the sections that follow. In each case, the grammar rule that is being applied is given, and an explanation of what program actions take place for that step has been included. In describing the actions taken for each parse step there has been no attempt to describe how the symbol table is constructed or how the values are preserved on the stack. The intent of this section is to describe what information needs to be retained and at what point in the parse it can be determined. Where no action is required for a given statement, or where the only action is to save the contents of the top of the stack, no explanation is given. Questions regarding the actual manipulation of information should be resolved by consulting the programs.


```

1  <program> ::= <id-div> <e-div> <d-div> PROCEDURE
    Reading the word PROCEDURE terminates the first part
    of the compiler.

2  <id-div> ::= IDENTIFICATION DIVISION.  PROGRAM-ID.
        <comment> . <auth> <date> <sec>

2  <auth> ::= AUTHOR . <comment> .
4          | <empty>
5  <date> ::= DATE-WRITTEN . <comment> .
6          | <empty>
7  <sec>  ::= SECURITY . <comment> .
8          | <empty>
9  <comment> ::= <input>
10         | <comment> <input>
11  <e-div> ::= ENVIRONMENT DIVISION . CONFIGURATION SECTION.
        <scr-obj> <i-o>
12  <src-obj> ::= SOURCE-COMPUTER . <comment> <debug> .
        OBJECT-COMPUTER . <comment> .
13  <debug> ::= DEBUGGING MODE
    Set a scanner toggle so that debug lines will be
    read.
14         | <empty>
15  <i-o>  ::= INPUT-OUTPUT SECTION . FILE-CONTROL .
        <file-control-list> <id<
16         | <empty>
17  <file-control-list> ::= <file-control-entry>
18         | <file-control-list> <file-
            control-entry>

```


19 <file-control-entry> ::= SELECT <id> <attribute-list> .

At this point all of the information about the file has been collected and the type of the file can be determined. File attributes are checked for compatibility and entered in the symbol table.

20 <attribute-list> ::= <one attrib>

21 | <attribute-list> <one attrib>

22 <one-attrib> ::= ORGANIZATION <org-type>

23 | ACCESS <acc-type> <relative>

24 | ASSIGN <input>

A file control block is built for the file using an INT operator.

25 <org-type> ::= SEQUENTIAL

No information needs to be stored since the default file organization is sequential.

26 | RELATIVE

The relative attribute is saved for production 19.

27 <acc-type> ::= SEQUENTIAL

This is the default.

28 | RANDOM

The random access mode needs to be saved for production 19.

29 <relative> ::= RELATIVE <id>

The pointer to the identifier will be retained by the current symbol pointer, so this production only saves a flag on the stack indicating that the production did occur.


```

30         | <empty>
31 <id> ::= I-O-CONTROL . <same-list>
32         | <empty>
33 <same-list> ::= <same-element>
34         | <same-list> <same-element>
35 <same-element> ::= SAME <id-string> .
36 <id-string> ::= <id>
37         | <id-string> <id>
38 <d-div> ::= DATA DIVISION . <file-section> <work> <link>
39 <file-section> ::= FILE SECTION . <file-list>

```

Actions will differ in production 64 depending upon whether this production has been completed. A flag needs to be set to indicate completion of the file section.

```

40         | <empty>
    The flag, indicated in production 39, is set.
41 <file-list> ::= <file-element>
42         | <file-list> <file-element>
43 <files> ::= FD <id> <file-control> . <record-description>

```

This statement indicates the end of a record description, and the length of the record and its address can now be loaded into the symbol table for the file name.

```

44 <file-control> ::= <file-list>
45         | <empty>
46 <file-list> ::= <file-element>
47         | <file-list> <file-element>

```


48 <file-element> ::= BLOCK <integer> RECORDS

49 | RECORD <rec-count>

The record length can be saved for comparison with the calculated length from the picture clauses.

50 | LABEL RECORDS STANDARD

51 | LABEL RECORDS OMITTED

52 | VALUE OF <id-string>

53 <rec-count> ::= <integer>

54 | <integer> TO <integer>

The TO option is the only indication that the file will be variable length. The maximum length must be saved.

55 <work> ::= WORKING-STORAGE SECTION . <record-description>

56 | <empty>

57 <link> ::= LINKAGE SECTION . <record-description>

58 | <empty>

59 <record-description> ::= <level-entry>

60 | <record-description> <level-entry>

61 <level-entry> ::= <integer> <data-id> <redefines>

 <data-type> .

The level entry needs to be loaded into the level stack. The level stack is used to keep track of the nesting of field definitions in a record. At this time there may be no information about the length of the item being defined, and its attributes may depend entirely upon its constituent fields. If there is a pending literal, the stack level to which it applies

is saved.

62 <data-id> ::= <id>

63 | FILLER

An entry is built in the symbol table to record information about this record field. It cannot be used explicitly in a program because it has no name, but its attributes will need to be stored as part of the total record.

64 <redefines> ::= REDEFINES <id>

The redefines option gives new attributes to a previously defined record area. The symbol table pointer to the area being redefined is saved so that information can be transferred from one entry to the other. In addition to the information saved relative to the redefinition, it is necessary to check to see if the current level number is less than or equal to the level recorded on the top of the level stack. If this is true, then all information for the item on the top of the stack has been saved and the stack can be reduced.

65 | <empty>

As in production 64, the stack is checked to see if the current level number indicates a reduction of the level stack. In addition, special action needs to be taken if the new level is 01. If an 01 level is encountered at this production prior to production 39 or 40 (the end of the file area), it is an implied

redefinition of the previous 01 level. In the working storage section, it indicates the start of a new record.

66 <data-type> ::= <prop-list>

67 | <empty>

68 <prop-list> ::= <data-element>

69 | <prop-list> <data-element>

70 <data-element> ::= PIC <input>

The <input> at this point is the character string that defines the record field. It is analyzed and the extracted information is stored in the symbol table.

71 | USAGE COMP

The field is defined to be a packed numeric field.

72 | USAGE DISPLAY

The DISPLAY format is the default, and thus no special action occurs.

73 | SIGN LEADING <separate>

This production indicates the presence of a sign in a numeric field. The sign will be in a leading position. If the <separate> indicator is true, then the length will be one longer than the picture clause, and the type will be changed.

74 | SIGN TRAILING <separate>

The same information required by production 73 must be recorded, but in this case the sign is trailing rather than leading.

75 | OCCURS <integer>

The type must be set to indicate multiple occurrences, and the number of occurrences saved for computing the space defined by this field.

76 | SYNC <direction>

Synchronization with a natural boundary is not required by this machine.

77 | VALUE <literal>

The field being defined will be assigned an initial value determined by the value of the literal through the use of an INT operator. This is only valid in the WORKING-STORAGE SECTION.

78 <direction> ::= LEFT

79 | RIGHT

80 | <empty>

81 <separate> ::= SEPARATE

The separate sign indicator is set on.

82 | <empty>

83 <literal> ::= <input>

The input string is checked to see if it is a valid numeric literal, and if valid, it is stored to be used in a value assignment.

84 | <lit>

This literal is a quoted string.

85 | ZERO

As is the case of all literals, the fact that there is a pending literal needs to be saved. In this case and the three following cases, an indicator of which

literal constant is being saved is all that is required. The literal value can be reconstructed later.

86 | SPACE

87 | QUOTE

88 <integer> ::= <input>

The input string is converted to an integer value for later internal use.

89 <id> ::= <input>

The input string is the name of an identifier and is checked against the symbol table. If it is in the symbol table, then a pointer to the entry is saved. If it is not in the symbol table, then an entry is added and the address of that entry is saved.

E. PART TWO

The second part includes all of the PROCEDURE DIVISION, and is the part where code generation takes place. As in the case of the first part, there was no intent to show how various pieces of information were retrieved but only what information was used in producing the output code.

1 <p-div> ::= PROCEDURE DIVISION <using> .

 <proc-body> EOF

This production indicates termination of the compilation. If the program has sections, then it will be necessary to terminate the last section with a RET 0 instruction. The code will be ended by the output of a TER operation.

2 <using> ::= USING id-string

Not implemented.

3 | <empty>

4 <id-string> ::= <id>

The identifier stack is cleared and the symbol table address of the identifier is loaded into the first stack location.

5 | <id-string> <id>

The identifier stack is incremented and the symbol table pointer stacked.

6 <proc-body> ::= <paragraph>

7 | <proc-body> <paragraph>

8 <paragraph> ::= <id> . <sentence-list>

The starting and ending address of the paragraph are entered into the symbol table. A return is emitted as the last instruction in the paragraph (RET 0).. When the label is resolved, it may be necessary to produce a BST operation to resolve previous references to the label.

9 | <id> SECTION .

The starting address for the section is saved. If it is not the first section, then the previous section ending address is loaded and a return (RET 0) is output. As in production 8, a BST may be produced.

10 <sentence-list> ::= <sentence>.

11 | <sentence-list> <sentence> .

12 <sentence> ::= <imperative>

13 | <conditional>

14 | ENTER <id> <opt-id>

This construct is not implemented. An ENTER allows statements from another language to be inserted in the source code.

15 <imperative> ::= ACCEPT <subid>

ACC <address> <length>

16 | <arithmetic>

17 | CALL <lit> <using>

This is not implemented.

18 CLOSE id

CLS file control block address

19 | <file-act>

20 | DISPLAY <lit/id> <opt-lit/id>

The display operator is produced for the first literal or identifier (DIS <address> <length>). If the second value exists, the same code is also produced for it.

21 | EXIT <program-id>

RET 0

22 | GO <id>

BRN <address>

23 | GO <id-string> DEPENDING <id>

GDP is output, followed by a number of parameters:

<the number of entries in the identifier stack> <the length of the depending identifier> <the address of

the depending identifier> <the address of each
identifier in the stack>.

24 | MOVE <lit/id> TO <subid>

The types of the two fields determine the move that is generated. Numeric moves go through register two using a load and a store. Non-numeric moves depend upon the result field and may be either MOV, MED or MNE. Since all of these instructions have long parameter lists, they have not been listed in detail.

25 | OPEN <type-action> <id>

This produces either OPN, OP1, or OP2 depending upon the <type-action>. Each of these is followed by a file control block address.

26 | PERFORM <id> <thru> <finish>

The PER operation is generated followed by the <branch address> <the address of the return statement to be set> and <the next instruction address>.

27 | <read-id>

28 | STOP <terminate>

If there is a terminate message, then STD is produced followed by <message address> <message length>. Otherwise STP is emitted.

29 <conditional> ::= <arithmetic> <size-error> <imperative>

A BST operator is output to complete the branch around the imperative from production 65.

30 | <file-act> <invalid> <imperative>

A BST operator is output to complete the branch from production 64.

31 | <if-nonterminal> <condition> <action>
 ELSE <imperative>

NEG will be emitted unless <condition> is a "NOT <cond-type>", in which case the two negatives will cancel each other.

Two BST operators are required. The first fills in the branch to the ELSE action. The second completes the branch around the <imperative> which follows ELSE.

32 | <read-id> <special> <imperative>

A BST is produced to complete the branch around the <imperative>.

33 <Arithmetic> ::= ADD <l/id> <opt-l/id> TO <subid> <round>

The existence of multiple load and store instructions make it difficult to indicate exactly what code will be generated for any of the arithmetic instructions. The type of load and store will depend on the nature of the number involved, and in each case the standard parameters will be produced. This parse step will involve the following actions: first, a load will be emitted for the first number into register zero. If there is a second number, then a load into register one will be produced for it, followed by an ADD and a STI. Next a load into register one will be generated for the result number. Then an ADD instruction will

load instructions and a SUB will also be emitted if
one of the register comparisons is required.

```
41 <cond-type> ::= NUMERIC
42             | ALPHABETIC
43             | <compare> <lit/id>
44 <not> ::= NOT
```

NEG is emitted unless the NOT is
part of an IF statement in which case the NEG in
the IF statement is cancelled.

```
45             | <empty>
46 <compare> ::= GREATER
47             | LESS
48             | EQUAL
49 <ROUND> ::= ROUNDED
50             | <empty>
51 <terminate> ::= <literal>
52             | RUN
53 <special> ::= <invalid>
54             | END
```

An ERO operator is emitted followed by a zero. The
zero acts as a filler in the code and will be back-
stuffed with a branch address. In this production and
several of the following, there is a forward branch on
a false condition past an imperative action. For an
example of the resolution, examine production 32.

```
55 <opt-id> ::= <subid>
56             | empty
```



```

57 <action> ::= <imperative>
    BRN 0
58         | NEXT SENTENCE
    BRN 0
59 <thru> ::= THRU <id>
60         | empty
61 <finish> ::= <l/id> TIMES
    LDI <address> <length> DEC 0
62         | UNTIL <condition>
63         | empty
64 <invalid> ::= INVALID
    INV 0
65 <size-error> ::= SIZE ERROR
    SER 0
66 <special-act> ::= <when> ADVANCING <how-many>
67         | <empty> ,
68 <when> ::= BEFORE
69         | AFTER
70 <how-many> ::= <integer>
71         | PAGE
72 <type-action> ::= INPUT
73         | OUTPUT
74         | I-O
75 <subid> ::= <subscript>
76         | id
77 <integer> ::= <input>

```


The identifier is checked against the symbol table, if it is not present, it is entered as an unresolved label.

79 <l/id> ::= <input>

The input value may be a numeric literal. If so, it is placed in the constant area with an INT operand. If it is not a numeric literal, then it must be an identifier, and it is located in the symbol table.

80 | <subscript>

81 | ZERO

82 <subscript> ::= <id> (<input>)

If the identifier was defined with a USING option, then the input string is checked to see if it is a number or an identifier. If it is an identifier, then an SCR operator is produced.

83 <opt-l/id> ::= <l/id>

84 | <empty>

85 <nn-lit> ::= <lit>

The literal string is placed into the constant area using an INT operator.

86 | SPACE

87 | QUOTE

88 <literal> ::= <nn-lit>

89 | <input>

The input value must be a numeric literal to be valid and is loaded into the constant area using an INT.

90 | ZERO


```
91 <opt-lit/id> ::= <lit/id>
94           | <empty>
95 <program-id> ::= <id>
96           | <empty>
97 <read-id> ::= READ <id>
```

There are four read operations: RDF, RVL, RRS, and RRR.

```
98 <if-nonterminal> ::= IF
```

The intermediate code file is the only product of the compiler that is retained. All of the needed information has been extracted from the symbol table, and it is not required by the interpreter. The intermediate code file can be examined through the use of the DECODE Program which translates the output file into a listing of mnemonics followed by the parameters.

APPENDIX A

MICRO-COBOL USER'S MANUAL

TABLE OF CONTENTS

I.	ORGANIZATION -----	47
II.	MICRO-COBOL ELEMENTS -----	48
III.	COMPILER PARAMETERS -----	84
IV.	RUN TIME CONVENTIONS -----	85
V.	FILE INTERACTIONS WITH CP/M -----	86
VI.	ERROR MESSAGES -----	88
	A. COMPILER FATAL MESSAGES -----	88
	B. COMPILER WARNINGS -----	88
	C. INTERPRETER FATAL ERRORS -----	90
	D. INTERPRETER WARNING MESSAGES -----	91

I. ORGANIZATION

The MICRO-COBOL compiler is designed to run on an 8080 system in an interactive mode, and requires at least 16K of RAM memory along with a diskette storage device. The compiler is composed of two parts, each of which reads a portion of the input file. Part one reads the input program and builds the symbol table. At the end of the Data Division, part one is overlayed by part two which uses the symbol table and the Procedure Division of the source program to produce the intermediate code which is written to the diskette as it is generated.

The BUILD Program reads the intermediate code file and creates the executable code memory image which is used by the interpreter. After the memory image has been created, the BUILD Program loads and passes control to the interpreter which then executes the intermediate code. .

II. MICRO-COBOL ELEMENTS

The procedure to compile and execute a MICRO-COBOL source program is covered in the next section. This section contains a description of each element in the language and shows simple examples of its use. The following conventions are used in explaining the formats: elements enclosed in broken braces < > are themselves complete entities and are described elsewhere in the manual. Elements enclosed in braces { } are choices, one of the elements which is to be used. Elements enclosed in brackets [] are optional. All elements in capital letters are reserved words and must be spelled exactly.

User names are indicated as lower case. These names have been restricted to 12 characters in length. The HYPO-COBOL specification requires that each name start with a letter. There are no restrictions in MICRO-COBOL on what characters must be in any position of a user name. However, it is generally good practice to avoid the use of number strings as names, since they will be taken as literal numbers wherever the context allows it. For example a record could be defined in the Data Division with the name 1234, but the command MOVE 1234 TO RECORD1 would result in the movement of the literal number not the data stored.

The input to the compiler does not need to conform to standard COBOL format. Free form input will be accepted as the default condition. If desired, sequence numbers can be entered in the first six positions of each line. When sequence numbers are used, a compiler parameter must set to cause the compiler to ignore them.

IDENTIFICATION DIVISION

ELEMENT:

IDENTIFICATION DIVISION Format

FORMAT:

IDENTIFICATION DIVISION.

PROGRAM-ID. <comment>.

[AUTHOR. <comment>.]

[DATA-WRITTEN. <comment>.]

[SECURITY. <comment>.]

DESCRIPTION:

This division provides information for program identification for the reader. The order of the lines is fixed.

EXAMPLES:

IDENTIFICATION DIVISION.

PROGRAM-ID. SAMPLE.

AUTHOR. PHIL MYLET.

ENVIRONMENT DIVISION

ELEMENT:

ENVIRONMENT DIVISION Format

FORMAT:

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. <comment> [DEBUGGING MODE].

OBJECT-COMPUTER. <comment>.

[INPUT-OUTPUT SECTION.

FILE-CONTROL.

<file-control-entry> . . .

[I-O-CONTROL.

SAME file-name-1 file-name-2 [file-name-3]

[file-name-4] [file-name-5].]]

DESCRIPTION:

This division determines the external nature of a file. In the case of CP/M all of the files used can be accessed either sequentially or randomly except for variable length files which are sequential only. The debugging mode is also set by this section.

<file-control-entry>

ELEMENT:

<file-control-entry>

FORMAT:

1.

```
SELECT file-name
      ASSIGN implementor-name
      [ORGANIZATION SEQUENTIAL]
      [ACCESS SEQUENTIAL].
```

2.

```
SELECT file-name
      ASSIGN implementor-name
      ORGANIZATION RELATIVE
      [ACCESS {SEQUENTIAL [RELATIVE data-name]}].
      {RANDOM RELATIVE data-name      }
```

DESCRIPTION:

The file-control-entry defines the type of file that the program expects to see. There is no difference on the diskette, but the type of reads and writes that are performed will differ. For CP/M the implementor name needs to conform to the normal specifications.

EXAMPLES:

1.

```
SELECT CARDS
      ASSIGN CARD.FIL.
```


2.

SELECT RANDOM-FILE

ASSIGN A.RAN

ORGANIZATION RELATIVE

ACCESS RANDOM RELATIVE RAND-FLAG.

ELEMENT:

DATA DIVISION Format

FORMAT:

DATA DIVISION.

[FILE SECTION.

[FD file-name

[BLOCK integer-1 RECORDS]

[RECORD [integer-2 TO] integer-3]

[LABEL RECORDS {STANDARD}]

{OMITTED}

[VALUE OF implementor-name-1 literal-1

[implementor-name-2 literal-2] ...].

[record-description-entry] ...] ...

[WORKING-STORAGE SECTION.

[<record-description-entry>] ...]

[LINKAGE SECTION.

[<record-description-entry>] ...]

DESCRIPTION:

This is the section that describes how the data is structured. There are no major differences from standard COBOL except for the following: 1. Label records make no sense on the diskette so no entry is required. 2. The VALUE OF clause likewise has no meaning for CP/M. 3. The linkage section has not been implemented.

If a record is given two lengths as in RECORD 12 TO 128, the file is taken to be variable length and can only be accessed in the sequential mode. See the section on files for more information.

<comment>

ELEMENT:

<comment>

FORMAT:

any string of characters

DESCRIPTION:

A comment is a string of characters. It may include anything other than a period followed by a blank or a reserved word, either of which terminate the string. Comments may be empty if desired, but the terminator is still required by the program.

EXAMPLES:

this is a comment

anotheroneallruntogether

8080b 16K

<data-description-entry>

ELEMENT:

<data-description-entry> Format

FORMAT:

```
level-number {data-name}  
           {FILLER}  
[REDEFINES data-name]  
[PIC character-string]  
[USAGE {COMP}    ]  
           {DISPLAY}  
[SIGN {LEADING} [SEPARATE]]  
           {TRAILING}  
[OCCURS integer]  
[SYNC [LEFT ]]  
           [RIGHT]  
[VALUE literal].
```

DESCRIPTION:

This statement describes the specific attributes of the data. Since the 8080 is a byte machine, there was no meaning to the SYNC clause, and thus it has not been implemented.

EXAMPLES:

01 CARD-RECORD.

02 PART PIC X(5).

02 NEXT-PART PIC 99V99 USAGE COMP.

02 FILLER.

03 NUMB PIC S9(3)V9 SIGN LEADING SEPARATE.

03 LONG-NUMB 9(15).

03 STRING REDEFINES LONG-NUMB PIC X(15).

02 ARRAY PIC 99 OCCURS 100.

PROCEDURE DIVISION

ELEMENT:

PROCEDURE DIVISION Format

FORMAT:

1.

PROCEDURE DIVISION [USING name1 [name2] ... [name5]].

section-name SECTION.

[paragraph-name. <sentence> [<sentence> ...] ...] ...

2.

PROCEDURE DIVISION [USING name1 [name2] ... [name5]].

paragraph-name. <sentence> [<sentence> ...] ...

DESCRIPTION:

As is indicated, if the program is to contain sections, then the first paragraph must be in a section. The USING option is part of the inter-program communication module and has not been implemented.

<sentence>

ELEMENT:

<sentence>

FORMAT:

<imperative-statement>

<conditional-statement>

ENTER verb

DESCRIPTION:

All sentences other than ENTER fall in one of the two main categories. ENTER is part of the inter-program communication module.

<imperative-statement>

ELEMENT:

<imperative-statement>

FORMAT

The following verbs are always imperatives:

ACCEPT

CALL

CLOSE

DISPLAY

EXIT

GO

MOVE

OPEN

PERFORM

STOP

The following may be imperatives:

arithmetic verbs without the SIZE ERROR statement

and DELETE, WRITE, and REWRITE without the INVALID

option.

<conditional-statements>

ELEMENT:

<conditional-statements>

FORMAT:

IF

READ

arithmetic verbs with the SIZE ERROR statement
and DELETE, WRITE, and REWRITE with the INVALID
option.

ACCEPT

ELEMENT:

ACCEPT

FORMAT:

ACCEPT <identifier>

DESCRIPTION:

This statement reads up to 72 characters from the console. The usage of the item must be DISPLAY.

EXAMPLES:

ACCEPT IMAGE

ACCEPT NUM(9)

ADD

ELEMENT:

ADD

FORMAT:

ADD {identifier} [{identifier-1}] TO identifier-2
 {literal} {literal} }
 [ROUNDED] [SIZE ERROR <imperative-statement>]

DESCRIPTION:

This instruction adds either one or two numbers to a third with the result being placed in the last location.

EXAMPLES:

ADD 10 TO NUMB1

ADD X Y TO Z ROUNDED.

ADD 100 TO NUMBER SIZE ERROR GO ERROR-LOC

CALL

ELEMENT:

CALL

FORMAT:

CALL literal [USING name1 [name2] ... [name5]

DESCRIPTION:

CALL is not implemented.

CLOSE

ELEMENT:

CLOSE

FORMAT:

CLOSE file-name

DESCRIPTION:

Files must be closed if they have been written.

However, the normal requirement to close an input
file prior to the end of processing does not exist.

EXAMPLES:

CLOSE FILE1

CLOSE RANDFILE

DELETE

ELEMENT:

DELETE

FORMAT:

DELETE record-name [INVALID <imperative-statement>]

DESCRIPTION:

This statement requires the record name, not the file name as in the standard form of the statement. Since there is no deletion mark in CP/M, this would normally result in the record still being readable. It is, therefore, filled with zeroes to indicate that it has been removed.

EXAMPLES:

DELETE RECORD1

DISPLAY

ELEMENT:

DISPLAY

FORMAT:

```
DISPLAY {identifier} [{identifier-1}]  
        {literal    } {literal    }
```

DESCRIPTION:

This displays the contents of an identifier or displays a literal on the console. Usage must be DISPLAY. The maximum length of the display is 72 positions.

EXAMPLES:

```
DISPLAY MESSAGE-1  
DISPLAY MESSAGE-3 10  
DISPLAY 'THIS MUST BE THE END'
```


DIVIDE

ELEMENT:

DIVIDE

FORMAT:

```
DIVIDE {identifier} INTO identifier-1 [ROUNDED]
      {literal   }
      [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The result of the division is stored in identifier-1;
any remainder is lost.

EXAMPLES:

```
DIVIDE NUMB INTO STORE
DIVIDE 25 INTO RESULT
```


EXIT

ELEMENT:

EXIT

FORMAT:

EXIT [PROGRAM]

DESCRIPTION:

The EXIT command causes no action by the interpreter but allows for an empty paragraph for the construction of a common return point. The optional PROGRAM statement is not implemented as it is part of the inter-program communication module.

EXAMPLES:

RETURN.

EXIT.

ELEMENT:

GO

FORMAT:

1.

GO procedure-name

2.

GO procedure-1 [procedure-2] ... procedure-20

DEPENDING identifier

DESCRIPTION:

The GO command causes an unconditional branch to the routine specified. The second form causes a forward branch depending on the value of the contents of the identifier. The identifier must be a numeric integer value. There can be no more than 20 procedure names.

EXAMPLES:

GO READ-CARD.

GO READ1 READ2 READ3 DEPENDING READ-INDEX.

IF

ELEMENT:

IF

FORMAT:

IF <condition> {imperative} ELSE imperative-2
 {NEXT SENTENCE}

DESCRIPTION:

This is the standard COBOL IF statement. Note that there is no nesting of IF statements allowed since the IF statement is a conditional.

EXAMPLES:

IF A GREATER B ADD A TO C ELSE GO ERROR-ONE.

IF A NOT NUMERIC NEXT SENTENCE ELSE MOVE ZERO TO A.

MOVE

ELEMENT:

MOVE

FORMAT:

MOVE {identifier-1} TO identifier-2
 {literal }

DESCRIPTION:

The standard list of allowable moves applies to this action. As a space saving feature of this implementation, all numeric moves go through the accumulators. This makes numeric moves slower than alphanumeric moves, and where possible they should be avoided. Any move that involves picture clauses that are exactly the same can be accomplished as an alphanumeric move if the elements are redefined as alphanumeric; also all group moves are alphanumeric.

EXAMPLES:

MOVE SPACE TO PRINT-LINE.

MOVE A(10) TO B(PTR).

MULTIPLY

ELEMENT:

MULTIPLY

FORMAT:

```
MULTIPLY {identifier} BY identifier-2 [ROUNDED]
        {literal    }
        [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The multiply routine requires enough space to calculate the result with the full number of decimal digits prior to moving the result into identifier-2. This means that a number with 5 places after the decimal multiplied by a number with 6 places after the decimal will generate a number with 11 decimal places which would overflow if there were more than 7 digits before the decimal place.

EXAMPLES:

MULTIPLY X BY Y.

MULTIPLY A BY B(7) SIZE ERROR GO OVERFLOW.

ELEMENT:

OPEN

FORMAT:

```
OPEN {INPUT file-name }  
      {OUTPUT file-name}  
      {I-O file-name  }
```

DESCRIPTION:

All three types of OPENs have the same effect on the diskette. However, they do allow for internal checking of the other file actions. For example, a write to a file set open as input will cause a fatal error.

EXAMPLES:

OPEN INPUT CARDS.

OPEN OUTPUT REPORT-FILE.

PERFORM

ELEMENT:

PERFORM

FORMAT

1.

PERFORM procedure-name [THRU procedure-name-2]

2.

PERFORM procedure-name [THRU procedure-name-2]

{identifier} TIMES

{integer }

3.

PERFORM procedure-name [THRU procedure-name-2]

UNTIL <condition>

DESCRIPTION:

All three options are supported. Branching may be either forward or backward, and the procedures called may have perform statements in them as long as the end points do not coincide or overlap.

EXAMPLES:

PERFORM OPEN-ROUTINE.

PERFORM TOTALS THRU END-REPORT.

PERFORM SUM 10 TIMES.

PERFORM SKIP-LINE UNTIL PG-CNT GREATER 60.

READ

ELEMENT:

READ

FORMAT:

1.

READ file-name INVALID <imperative-statement>

2.

READ file-name END <imperative-statement>

DESCRIPTION:

The invalid condition is only applicable to files in a random mode. All sequential files must have an END statement.

EXAMPLES:

READ CARDS END GO END-OF-FILE.

READ RANDOM-FILE INVALID MOVE SPACES TO REC-1.

REWRITE

ELEMENT:

REWRITE

FORMAT:

REWRITE file-name [INVALID <imperative>]

DESCRIPTION:

REWRITE is only valid for files that are open in the 1-0 mode. The INVALID clause is only valid for random files. This statement results in the current record being written back into the place that it was just read from. Note that this requires a file name not a record name.

EXAMPLES:

REWRITE CARDS.

REWRITE RAND-1 INVALID PERFORM ERROR-CHECK.

STOP

ELEMENT:

STOP

FORMAT:

STOP {RUN }
 {literal}

DESCRIPTION:

This statement ends the running of the interpreter.
If a literal is specified, then the literal is
displayed on the console prior to termination of
the program.

EXAMPLES:

STOP RUN.

STOP 1.

STOP "INVALID FINISH".

SUBTRACT

ELEMENT:

SUBTRACT

FORMAT:

```
SUBTRACT {identifier-1} [identifier-2] FROM identifier-3
        {literal-1    } [literal-2    ]
        [ROUNDED] [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

Identifier-3 is decremented by the value of identifier/literal one, and, if specified, identifier/literal two. The results are stored back in identifier-3. Rounding and size error options are available if desired.

EXAMPLES:

SUBTRACT 10 FROM SUB(12).

SUBTRACT A B FROM C ROUNDED.

WRITE

ELEMENT:

WRITE

FORMAT:

1.

```
WRITE file-name [{BEFORE} ADVANCING {INTEGER}]
                  {AFTER }           {PAGE   }
```

2.

```
WRITE file-name INVALID <imperative-statement>
```

DESCRIPTION:

There is no printer on the 8080 system here, so the ADVANCING option is not implemented. The INVALID option only applies to random files.

EXAMPLES:

```
WRITE OUT-FILE.
```

```
WRITE RAND-FILE INVALID PERFORM ERROR-RECOV.
```


<condition>

ELEMENT:

<condition>

FORMAT:

RELATIONAL CONDITION:

```
{identifier-1} [NOT] {GREATER} {identifier-2}
{literal-1   }      {LESS   } {literal-2   }
               {EQUAL   }
```

CLASS CONDITION:

```
identifier [NOT] {NUMERIC   }
               {ALPHABETIC}
```

DESCRIPTION:

It is not valid to compare two literals. The class condition NUMERIC will allow for a sign if the identifier is signed numeric.

EXAMPLES:

```
A NOT LESS 10.
LINE GREATER "C".
NUMB1 NOT NUMERIC.
```


Subscripting

ELEMENT:

Subscripting

FORMAT:

data-name (subscript)

DESCRIPTION:

Any item defined with an OCCURS may be referenced by a subscript. The subscript may be a literal integer, or it may be a data item that has been specified as an integer. If the subscript is signed, the sign must be positive at the time of its use.

EXAMPLES:

A(10)

ITEM(SUB)

III. COMPILER PARAMETERS

There are four compiler parameters which are controlled by entries on the first line of the program. A parameter consists of a dollar sign followed by a letter.

\$L -- list the input code on the screen as the program is compiled. Default is on. Error messages will be difficult to understand with this parameter turned off, but it may be desirable when used with a slow output device.

\$S -- sequence numbers are in the first six positions of each record. Default is off.

\$P -- list productions as they occur. Default is off.

\$T -- list tokens from the scanner. Default is off.

IV. RUN TIME CONVENTIONS

This section explains how to compile and execute MICRO-COBOL source programs. The compiler expects to see a file with a type of CBL as the input file. In general, the input is free form. If the input includes line numbers then the compiler must be notified by setting the appropriate parameter. The compiler is started by typing COBOL <file-name>. Where the file name is the system name of the input file. There is no interaction required to start the second part of the compiler. The output file will have the same file name as the input file, and will be given a file type of CIN. Any previous copies of the file will be erased.

The interpreter is started by typing EXEC <file-name>. The first program is a loader, and it will display "LOAD FINISHED" to indicate successful completion. The run-time package will be brought in by the build program, and execution should continue without interruption.

V. FILE INTERACTIONS WITH CP/M

The file structure that is expected by the program imposes some restrictions on the system. References 3 and 4 contain detailed information on the facilities of CP/M, and should be consulted for details. The information that has been included in this section is intended to explain where limitations exist and how the program interacts with the system.

All files in CP/M are on a random access device, and there is no way for the system to distinguish sequential files from files created in a random mode. This means that the various types of reads and writes are all valid to any file that has fixed length records. The restrictions of the ASSIGN statement do prevent a file from being open for both random and sequential actions during one program.

Each logical record is terminated by a carriage return and a line feed. In the case of variable length records, this is the only end mark that exists. This convention was adopted to allow the various programs which are used in CP/M to work with the files. Files created by the editor, for example, will generally be variable length files. This convention does remove the capability of reading variable length files in a random mode.

All of the physical records are assumed to be 128 bytes in length, and the program supplies buffer space for

records in addition to the logical records. Logical records may be of any desired length.

VI. ERROR MESSAGES

A. COMPILER FATAL MESSAGES

- BR Bad read -- disk error, no corrective action can
 be taken in the program.
- CL Close error -- unable to close the output file.
- MA Make error -- could not create the output file.
- MO Memory overflow -- the code and constants generated
 will not fit in the allotted memory space.
- OP Open error -- can not open the input file, or no
 such file present.
- ST Symbol table overflow -- symbol table is too large
 for the allocated space.
- WR Write error -- disk error, could not write a code
 record to the disk.

B. COMPILER WARNINGS

- EL Extra levels -- only 10 levels are allowed.
- FT File type -- the data element used in a read or
 write statement is not a file name.
- IA Invalid access -- the specified options are not an
 allowable combination.

ID Identifier stack overflow -- more than 20 items in
a GO TO -- DEPENDING statement.

IS Invalid subscript -- an item was subscripted but
it was not defined by an OCCURS.

IT Invalid type -- the field types do not match for
this statement.

LE Literal error -- a literal value was assigned to an
item that is part of a group item previously assigned
a value.

NF No file assigned -- there was no SELECT clause for
this file.

NI Not implemented -- a production was used that is
not implemented.

NN Non-numeric -- an invalid character was found in a
numeric string.

NP No production -- no production exists for the
current parser configuration; error recovery will
automatically occur.

NV Numeric value -- a numeric value was assigned to a
non-numeric item.

PC Picture clause -- an invalid character or set of
characters exists in the picture clause.

- PF Paragraph first -- a section header was produced after a paragraph header, which is not in a section.
- R1 Redefine nesting -- a redefinition was made for an item which is part of a redefined item.
- R2 Redefine length -- the length of the redefinition item was greater than the item that it redefined.
- SE Scanner error -- the scanner was unable to read an identifier due to an invalid character.
- SG Sign error -- either a sign was expected and not found, or a sign was present when not valid.
- SL Significance loss -- the number assigned as a value is larger than the field defined.
- TE Type error -- the type of a subscript index is not integer numeric.
- VE Value error -- a value statement was assigned to an item in the file section.

C. INTERPRETER FATAL ERRORS

- CL Close error -- the system was unable to close an output file.
- ME Make error - the system was unable to make an input file on the disk.

NF No file -- an input file could not be opened.

WI Write to input -- a write was attempted to an
 input file.

D. INTERPRETER WARNING MESSAGES

EM End mark -- a record that was read did not have a
 carriage return or a line feed in the expected
 location.

GD Go to depending -- the value of the depending
 indicator was greater than the number of available
 branch addresses.

IC Invalid character -- an invalid character was loaded
 into an output field during an edited move. For
 example, a numeric character into an alphabetic-only
 field.

SI Sign invalid -- the sign is not a "+" or a "-".

WR.

LIST OF REFERENCES

1. Mylet, P. R. MICRO-COBOL a subset of Navy Standard HYPO-COBOL for Micro-computers, Master's Thesis; Naval Postgraduate School, September 1978.
2. Craig, A. S. MICRO-COBOL an implementation of Navy Standard HYPO-COBOL for microprocessor-based computer systems, Master's Thesis, Naval Postgraduate School, March 1977.
3. Digital Research, An Introduction to CP/M Features and Facilities, 1976.
4. Digital Research, CP/M Interface Guide, 1976.
5. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.
6. Intel Corporation, 8080 Simulator Software Package, 1974.
7. Software Development Division, ADPE Selection Office, Department of the Navy, HYPO-COBOL, April 1975.

APPENDIX B

MICRO-COBOL FILE CREATION

The MICRO-COBOL compiler and interpreter source files currently exist in PLM80 and are edited and compiled under ISIS on the INTEL MDS System. This is a description of the procedure used to create the executable files required to compile and interpret MICRO-COBOL programs. The MICRO-COBOL compiler and interpreter run under CP/M by executing the following four object code files.

1. COBOL.COM
2. PART2.COM
3. EXEC.COM
4. INTERP.COM

These four files are created from the following six PLM80 source programs.

1. PART1.PLM
2. PART2.PLM
3. BUILD.PLM
4. INTERP.PLM
5. INTRDR.PLM
6. READER.PLM

The procedure used to create the four object files involves compiling, linking, and locating each of the six source files under ISIS. The DDT program is then used under CP/M to construct the executable files. Each of the

following steps describe the action to be taken and, where appropriate, the command string to be entered into the computer.

1. An ISIS system diskette containing the PLM80 compiler is placed into drive A and a non-system diskette containing the source programs is placed into drive B.

2. Compile the PLM source file under ISIS.

```
PLM80 :F1:<filename>.PLM DEBUG
```

DEBUG saves the symbol table and line files for later use during debugging sessions.

3. Link the PLM80 object file.

```
LINK :F1:<filename>.OBJ, TRINT.OBJ, PLM80.LIB TO  
:F1:<filename>.MOD
```

4. Locate object file.

```
LOCATE :F1:<filename>.MOD CODE(103H)
```

5. Replace ISIS system diskette in drive A with a CP/M system diskette and reboot the system.

6. Transfer the located ISIS file from the diskette in drive B to the CP/M diskette in drive A.

```
FROMISIS <filename>
```

7. Convert the ISIS file to CP/M executable form.

```
OBJCPM <filename>
```


At this point the object file is in machine readable form and will run under CP/M when called properly. INTERP.COM and PART2.COM are called by EXEC.COM and PART1.COM and need no further work. EXEC.COM and PART1.COM need to be constructed from the remaining four files.

EXEC.COM is created by entering the following commands under CP/M.

1. DDT BUILD.COM
2. IINTRDR.HEX
3. R1C00
4. AlCB5
5. JMP 5
6. AlCC1
7. JMP 5
8. CONTROL-C
9. SAVE 29 EXEC.COM

PART1.COM is created by entering the following commands under CP/M.

1. DDT PART1.COM
2. IREADER.HEX
3. RFB00 6200
4. AlF90
5. JMP 3100 0P00
6. Control-C
7. SAVE 44 COBOL.COM

MICRO-COBOL programs may now be executed in the following manner. The source program is named, <filename>.CBL. The command, "COBOL <filename>", causes the MICRO-COBOL source program, <filename>.CBL, to be read in from diskette and compiled. During the compile, the intermediate code file, <filename>.CIN, is written out to diskette as it is generated. The command, "EXEC <filename>", causes the file, <filename>.CIN, to be executed.

APPENDIX C

LIST OF INOPERATIVE CONSTRUCTS

The following is a list of MICRO-COBOL elements that were not implemented at the beginning of this project. In most cases code had been written to implement the element but it was either incomplete or incorrect. The elements marked with an asterisk still have bugs and need additional work.

MULTIPLY

<condition>

STOP <literal>

IF

PERFORM <procedure 1> THRU <procedure 2>

PERFORM <procedure> <n> TIMES

PERFORM <procedure> UNTIL <condition>

FILE I/O *

Numeric Edit *

The following HYPO-COBOL elements are part of MICRO-COBOL only to the extent that they are defined in the grammar. No code has been written to support them.

USING

CALL

ENTER

<when> ADVANCING <how-many>

It must be pointed out that this information is based only on informal testing with very simple programs. MICRO-COBOL is only now at a stage at which it is appropriate to conduct exhaustive testing using the HYPO-COBOL Compiler Validation System.

APPENDIX D

MICRO-COBOL PARSE TABLE GENERATION

The parse tables for MICRO-COBOL were generated on the IBM 360 using the LALR(1) parse table generator described in Reference 11. There are basically two steps involved in generating the tables. First, a deck of cards containing the grammar is entered into the computer using the following JCL:

```
//GO EXEC PGM= LALR,REGION=220K
//STEPLIB DD DSN=F0963.LALR,UNIT=2314,
          VOL=SER=LINDA,DISP=SHR
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FB,
          LRECL=133,BLKSIZE=3325),
//SPACE=(CYL,(1,1))
//NONTERM DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//FSMDATA DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//PTABLES DD SYSOUT=B,
          DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSIN DD *
```

The output from this run is a listing and deck containing the tables in XPL compatible format. This deck is then translated into PLM compatible format using the following JCL and an XPL program which is available in the card deck library in the Computer Science Department at the Naval Postgraduate School.


```
//EXEC XCOM  
//COMP.SYSIN DD *  
//GO.SYSPUNCH DD SYSOUT=B,  
        DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)  
//GO.SYSIN DD *
```

The tables are then transferred to a diskette and edited into the PLM80 source program using the ISIS COPY and EDIT features on the INTEL MDS System.

ISIS-II PLM-80 V3.1 COMPILATION OF MODULE READER
 OBJECT MODULE PLACED IN F1 READER OBJ
 COMPILER INVOKED BY PLM80 F1 READER PLM

```

1      * PAGELENGTH(30)
      READER.
      DO.

      /* COBOL COMPILER - PART 2 READER */

      /* THIS PROGRAM IS LOADED IN WITH THE PART 1 PROGRAM
      AND IS CALLED WHEN PART 1 IS FINISHED. THIS PROGRAM
      OPENS THE PART2.COM FILE THAT CONTAINS THE CODE FOR
      PART 2 OF THE COMPILER, AND READS IT INTO CORE. AT
      THE END OF THE READ OPERATION, CONTROL IS PASSED TO
      THE SECOND PART PROGRAM */

      /*          0100H:  LOAD POINT */

2      1      DECLARE

      START  LITERALLY 0100H. /* STARTING LOCATION FOR PASS 2 */
      ADDR   ADDRESS INITIAL(START).
      FCB(15) BYTE INITIAL(0, PASS2  COM, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0).
      I       ADDRESS.

3      1      MON1. PROCEDURE(F, A) EXTERNAL.
4      2      DECLARE F BYTE, A ADDRESS.
5      2      END MON1.

6      1      MON2. PROCEDURE(F, A) BYTE EXTERNAL.
7      2      DECLARE F BYTE, A ADDRESS.
8      2      END MON2.

9      1      BOOT. PROCEDURE EXTERNAL.
10     2      END.

11     1      OPEN. PROCEDURE (FCB) BYTE.
12     2      DECLARE FCB ADDRESS.
13     2      RETURN MON2 (15, FCB).
14     2      END.

15     1      READ. PROCEDURE (ADDR) BYTE.
16     2      DECLARE ADDR ADDRESS.
17     2      CALL MON1 (26, ADDR). /* SET DMA ADDRESS */
18     2      RETURN MON2 (20, FCB). /* READ, AND RETURN ERROR CODE */
19     2      END.

20     1      ERROR. PROCEDURE(CODE).
21     2      DECLARE CODE ADDRESS.
22     2      CALL MON1(2, HIGH(CODE)).
23     2      CALL MON1(2, LOW(CODE)).
24     2      CALL TIME(10).
25     2      CALL BOOT.
26     2      END ERROR.
27     1      CALL MON1 (26, 0100H).

      /* OPEN PASS2.COM */
28     1      IF OPEN(FCB) # 255 THEN CALL ERROR('02')
      /* READ IN FILE */

29     1      I = 0100H. /* INITIAL ADDRESS */
30     1      DO WHILE READ(I) # 0. /* READ 1 SECTOR */
31     2      I = I + 0020H. /* BUMP DMA ADDRESS */
32     2      END.
33     2      END.

34     1      CALL MON1 (26, 0020H). /* RESET DMA ADDRESS */
35     1      CALL ADDR.

36     1      END.

```

MODULE INFORMATION

```

CODE AREA SIZE      = 0030H    157D
VARIABLE AREA SIZE  = 001BH    43D
MAXIMUM STACK SIZE  = 0004H    4D
67 LINES READ
0 PROGRAM ERROR(S)

```

END OF PLM-80 COMPILATION

1375-II PL/M-80 V2.1 COMPILATION OF MODULE INTERP
 OBJECT MODULE PLACED IN F1 INTERP.OBJ
 COMPILER INVOKED BY PL/M80 F1 INTERP.PLM

```

1      #PAGELENGTH(30)
      INTERP:  /* NAME OF MODULE */
              DO,

              /* COSOL COMPILER - INTERP HEADER */

              /* THIS PROGRAM IS CALLED BY THE BUILD PROGRAM AFTER
              CINTERP.COM HAS BEEN OPENED, AND READS THE CODE INTO MEMORY.
              */

              * 30H - LOAD POINT */

2      1      DECLARE

              START LITERALLY '100H', /* STARTING LOCATION FOR PASS 2 */
              INTERP ADDRESS INITIAL(START),
              I ADDRESS INITIAL (0000H),

3      1      MONA PROCEDURE(F,A),
4      2      DECLARE F BYTE, A ADDRESS,
5      2      L.GO TO L /* PATCH TO -> "JMP 8005" */
6      2      END MONA,

7      1      MONB PROCEDURE(F,A,BYTE),
8      2      DECLARE F BYTE, A ADDRESS,
9      2      L.GO TO L /* PATCH TO -> "JMP 8005" */
10     2      RETURN 0; /* CAP -> "NO-OP" */
11     2      END MONB,

12     1      DO WHILE L,
13     2      CALL MONA (26, (I:=I+0000H)), /* SET DMA ADDRESS */
14     2      IF MONB (20, 50H) (0) 0 THEN
15     1      CALL INTERP,
16     1      END,
17     1      END,

```

MODULE INFORMATION.

```

CODE AREA SIZE      = 0047H      71D
VARIABLE AREA SIZE = 0000H      10D
MAXIMUM STACK SIZE = 0002H      2D
16 LINES READ
0 PROGRAM ERRORS,

```

END OF PL/M-80 COMPILATION

ISIS-II PL/M-80 V0.1 COMPILATION OF MODULE BUILD
 OBJECT MODULE PLACED IN F1 BUILD OBJ
 COMPILER INVOKED BY PLM80 F1 BUILD PLM

```

1      IFPAGELENGTH=800
      BUILD
      DO.
      /* NORMALLY ORG ED AT 100H */

      /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE CODEL COMPILER
      AND BUILDS THE ENVIRONMENT FOR THE CODEL INTERPRETER */

2      1      DECLARE

      LIT          LITERALLY      'LITERALLY',
      BOOT         LIT            '0?',
      EOS          LIT            '5?',
      TRUE         LIT            '1?',
      FALSE        LIT            '0?',
      FOREVER       LIT            'WHILE TRUE',
      FCB          ADDRESS        INITIAL (5CH),
      FCB#BYTE      BASED FCB      FCB BYTE,
      FCB#BYTE#A    BASED FCB (10) BYTE,
      I             BYTE,
      ADDR         ADDRESS        INITIAL (100H),
      CHAR         BASED ADDR      ADDR BYTE,
      BUFF#END      LIT            '100H',
      INTERP#FCB    (10)          BYTE INITIAL(0, 0, 0, 0, 0),
      CODE#IND#SET   BYTE          INITIAL (TRUE),
      READER#LOCATION LIT            '100H',
      INTERP#ADDRESS ADDRESS        INITIAL(2000H),
      INTERP#CONTENT BASED INTERP#ADDRESS ADDRESS,
      I#BYTE        BASED INTERP#ADDRESS (2) BYTE,
      CODE#CTR       ADDRESS,
      C#BYTE         BASED CODE#CTR BYTE,
      BASE           ADDRESS,
      B#ADDR         BASED BASE     BASE ADDRESS,
      B#BYTE         BASED BASE     BASE (4) BYTE,

2      1      MON1 PROCEDURE (F,A) EXTERNAL,
4      2      DECLARE F BYTE, A ADDRESS,
5      2      END MON1,

6      1      MON2 PROCEDURE (F,A) BYTE EXTERNAL,
7      2      DECLARE F BYTE, A ADDRESS,
8      2      END MON2,

9      1      PRINT#CHAR PROCEDURE(CHAR),
10     2      DECLARE CHAR BYTE,
11     2      CALL MON1(2,CHAR),
12     2      END PRINT#CHAR,

13     1      CRLF PROCEDURE,
14     2      CALL PRINT#CHAR(10),
15     2      CALL PRINT#CHAR(10),
16     2      END CRLF,

17     1      PRINT PROCEDURE(A),
18     2      DECLARE A ADDRESS,
19     2      CALL CRLF,
20     2      CALL MON1(9,A),
21     2      END PRINT,

22     1      OPEN PROCEDURE (A) BYTE,
23     2      DECLARE A ADDRESS,
24     2      RETURN MON2(10,A),
25     2      END OPEN,

26     1      RESBOOT PROCEDURE,
27     2      ADDR = BOOT, CALL ADDR,
28     2      END RESBOOT,

30     1      MOVE PROCEDURE(FROM, DEST, COUNT),
31     2      DECLARE (FROM, DEST, COUNT) ADDRESS,
32     2      IF BASED FROM, 0 BASED DEST, BYTE,
33     2      DO WHILE COUNT #COUNT-1 AND 0FFFFH,
34     3      D=F,
35     3      FROM=FROM+1,

```



```

35 1      DEST=DEST+1.
36 3      END.
37 2      END MOVE.

38 1      GET$CHAR. PROCEDURE BYTE.
39 2      IF (ADDR =ADDR + 1)/=BUFF$END THEN
40 3      DO.
41 4          IF NON2<20, FCB><08 THEN
42 5          DO.
43 6              CALL PRINT( ('END OF INPUT ',$?))
44 7              CALL REBOOT.
45 8          END.
46 9          ADDR=30H.
47 10      END.
48 2      RETURN CHAR.
49 2      END GET$CHAR.

50 1      NEXT$CHAR. PROCEDURE.
51 2      CHAR=GET$CHAR.
52 2      END NEXT$CHAR.

53 1      STORE. PROCEDURE<COUNT>.
54 2      DECLARE COUNT BYTE.
55 2      IF CODE$NOT$SET THEN
56 3      DO.
57 4          CALL PRINT( ('CODE ERROR$ ')).
58 5          CALL NEXT$CHAR.
59 6          RETURN.
60 7      END.
61 3      DO I=1 TO COUNT.
62 4          C$BYTE=CHAR.
63 5          CALL NEXT$CHAR.
64 6          CODE$CTR=CODE$CTR+1.
65 7      END.
66 2      END STORE.

67 1      BACK$STUFF. PROCEDURE.
68 2      DECLARE (HOLD, STUFF) ADDRESS.
69 3      BASE= HOLD.
70 3      DO I=0 TO 3.
71 4          B$BYTE(I)=GET$CHAR.
72 5      END.
73 2      DO FOREVER.
74 3          BASE=HOLD.
75 4          HOLD=B$ADDR.
76 5          B$ADDR=STUFF.
77 6          IF HOLD=0 THEN
78 7          DO.
79 8              CALL NEXT$CHAR.
80 9              RETURN.
81 10      END.
82 3      END.
83 2      END BACK$STUFF.

84 1      START$CODE. PROCEDURE.
85 2      CODE$NOT$SET=FALSE.
86 3      I$BYTE(0)=GET$CHAR.
87 4      I$BYTE(1)=GET$CHAR.
88 5      CODE$CTR=INTERP$CONTENT.
89 6      CALL NEXT$CHAR.
90 2      END START$CODE.

91 1      GO$DEPENDING. PROCEDURE.
92 2      CALL STORE(1).
93 3      CALL STORE(SHL$CHAR, 1) + 4).
94 2      END GO$DEPENDING.

95 1      INITIALIZE. PROCEDURE.
96 2      DECLARE (COUNT, WHERE, HOW$MANY) ADDRESS.
97 3      BASE= WHERE.
98 3      DO I=0 TO 3.
99 4          B$BYTE(I)=GET$CHAR.
100 5      END.
101 3      BASE=WHERE - 1.
102 2      DO COUNT = 1 TO HOW$MANY.
103 3          B$BYTE(COUNT)=GET$CHAR.
104 4      END.
105 2      CALL NEXT$CHAR.
106 2      END INITIALIZE.

```



```

107 1      BUILD PROCEDURE.
108 2      DECLARE
          F2 LIT '9',
          F3 LIT '9',
          F4 LIT '21',
          F5 LIT '24',
          F6 LIT '32',
          F7 LIT '39',
          F8 LIT '49',
          F10 LIT '54',
          F11 LIT '60',
          F13 LIT '61',
          GDP LIT '62',
          INT LIT '63',
          BST LIT '64',
          TER LIT '65',
          SCO LIT '66',

109 2      DO FOREVER,
110 3          IF CHAR < F2 THEN CALL STORE(1),
111 3          ELSE IF CHAR < F3 THEN CALL STORE(2),
112 3          ELSE IF CHAR < F4 THEN CALL STORE(3),
113 3          ELSE IF CHAR < F5 THEN CALL STORE(4),
114 3          ELSE IF CHAR < F6 THEN CALL STORE(5),
115 3          ELSE IF CHAR < F7 THEN CALL STORE(6),
116 3          ELSE IF CHAR < F8 THEN CALL STORE(7),
117 3          ELSE IF CHAR < F10 THEN CALL STORE(9),
118 3          ELSE IF CHAR < F11 THEN CALL STORE(10),
119 3          ELSE IF CHAR < F13 THEN CALL STORE(11),
120 3          ELSE IF CHAR < GDP THEN CALL STORE(12),
121 3          ELSE IF CHAR = GDP THEN CALL GO$DEPENDING,
122 3          ELSE IF CHAR = BST THEN CALL BACK$STUFF,
123 3          ELSE IF CHAR = INT THEN CALL INITIALIZE,
124 3          ELSE IF CHAR = TER THEN
125 3              DO,
126 3              CALL PRINT('LOAD FINISHED$'),
127 3              RETURN,
128 3          ENO,
129 3          ELSE IF CHAR = SCO THEN CALL START$CODE,
130 3          ELSE DO,
131 3              IF CHAR < 0FFH THEN CALL PRINT('LOAD ERROR$'),
132 3              CALL NEXT$CHAR,
133 3          ENO,
134 3      ENO,
135 2      ENO BUILD,

          /* PROGRAM EXECUTION STARTS HERE */

132 1      FCB$BYTE#A(32), FCB$BYTE=0,
133 1      CALL MOVE('CIN',0,0,0,0), FCB = 3,7),
134 1      IF OPEN(FCB)=255 THEN
135 1          DO,
136 2          CALL PRINT('FILE NOT FOUND $'),
137 2          CALL REBOOT,
138 2      ENO,
139 1      CALL NEXT$CHAR,
140 1      CALL BUILD,
141 1      CALL MOVE(INTERP$FCB,FCB,32),
142 1      FCB$BYTE#A(32) = 0,
143 1      IF OPEN(FCB)=255 THEN
144 1          DO,
145 2          CALL PRINT('INTERPRETER NOT FOUND $'),
146 2          CALL REBOOT,
147 2      ENO,
148 1      CALL MOVE(READER$LOCATION, 80H, 80H),
149 1      ADDR = 80H, CALL ADDR /* BRANCH TO 80H */
150 1      ENO,
151 1

```

MODULE INFORMATION

```

CODE AREA SIZE      = 0402H    1026D
VARIABLE AREA SIZE  = 0040H     67D
MAXIMUM STACK SIZE  = 0012H     18D
237 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-90 COMPILATION

[illegible]


```

52 2      DECLARE DCNT BYTE,
53 2      IF (DCNT = MON2(20, INPUT#FCB))=1 THEN CALL FATAL$ERROR('ER'),
54 2      RETURN NOT(DCNT),
55 2  END MORE$INPUT,

57 1  MAKE PROCEDURE,
    /* DELETES ANY EXISTING COPY OF THE OUTPUT FILE
       AND CREATES A NEW COPY */
58 2      CALL MON1(13, OUTPUT#FCB),
59 2      IF MON2(22, OUTPUT#FCB)=255 THEN CALL FATAL$ERROR('NA'),
60 2  END MAKE,

62 1  WRITE$OUTPUT PROCEDURE,
    /* WRITES OUT A BUFFER */
63 2      CALL MON1(26, OUTPUT#BUFF), /* SET DMA */
64 2      IF MON2(21, OUTPUT#FCB)=0 THEN CALL FATAL$ERROR('WR'),
65 2      CALL MON1(26, 60H), /* RESET DMA */
66 2  END WRITE$OUTPUT,

68 1  MOVE PROCEDURE(SOURCE, DESTINATION, COUNT),
    /* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
69 2      DECLARE (SOURCE, DESTINATION) ADDRESS,
    (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION, COUNT) BYTE,
70 2      DO WHILE (COUNT = COUNT - 1) < 255,
71 3          D$BYTE=S$BYTE,
72 3          SOURCE=SOURCE + 1,
73 3          DESTINATION = DESTINATION + 1,
74 3      END,
75 2  END MOVE,

76 1  FILL PROCEDURE(ADDR, CHAR, COUNT),
    /* MOVES CHAR INTO ADDR FOR COUNT BYTES */
77 2      DECLARE ADDR ADDRESS,
    (CHAR, COUNT, DEST BASED ADDR) BYTE,
78 2      DO WHILE (COUNT = COUNT - 1) < 255,
79 3          DEST=CHAR,
80 3          ADDR=ADDR + 1,
81 3      END,
82 2  END FILL,

/* * * * * * SCANNER LITS * * * * */
83 1  DECLARE
    LITERAL      LIT      '15',
    INPUT$STR     LIT      '32',
    PERIOD        LIT      '1',
    INVALID       LIT      '0',

/* * * * * * SCANNER TABLES * * * * */
84 1  DECLARE TOKEN$TABLE (*) BYTE DATA
    /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST RESERVED WORD
       FOR EACH LENGTH OF WORD */
    (0, 0, 1, 4, 5, 15, 22, 32, 36, 44, 47, 49, 51, 53, 56, 57),

    TABLE (**) BYTE DATA (FD, OF, TO, PIC, COMP, DATA, FILE,
    LEFT, MODE, SAME, SIGN, SYNC, ZERO, BLOCK, LABEL,
    QUOTE, RIGHT, SPACE, USAGE, VALUE, ACCESS, ASSIGN,
    AUTHOR, FILLER, OCCURS, RANDOM, RECORD, SELECT,
    DISPLAY, LEADING, LINKAGE, OMITTED, RECORDS,
    SECTION, DIVISION, RELATIVE, SECURITY, SEPARATE, STANDARD,
    TRAILING, DEBUGGING, PROCEDURE, REDEFINES,
    PROGRAM-ID, SEQUENTIAL, ENVIRONMENT, I-O-CONTROL,
    DATE-WRITTEN, FILE-CONTROL, INPUT-OUTPUT, ORGANIZATION,
    CONFIGURATION, IDENTIFICATION, OBJECT-COMPUTER,
    SOURCE-COMPUTER, WORKING-STORAGE),

    OFFSET (16) ADDRESS
    /* NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH LENGTH */
    INITIAL (0, 0, 0, 6, 9, 45, 88, 123, 170, 218, 245, 265,
    287, 325, 348, 362),

    WORD$COUNT (**) BYTE DATA
    /* NUMBER OF WORDS OF EACH SIZE */
    (0, 0, 3, 1, 5, 7, 3, 6, 6, 3, 2, 2, 4, 1, 1, 3),

    MAX$LEN      LIT      '16',
    ADD$END (**) BYTE DATA (PROCEDURE),
    LOOKED       BYTE      INITIAL (0),
    HOLD         BYTE,
    BUFFER$END    ADDRESS    INITIAL (100H),
    NEXT         BASED      POINTER BYTE,
    IN$BUFF       LIT      '60H',
    CHAR         BYTE,
    ACCUM$LENG    LIT      '50',
    ACCUM$LENG#1 LIT      '51', /* = TO ACCUM$LENG PLUS 1 */
    ACCUM (ACCUM$LENG#1) BYTE,

```



```

      DISPLAY(74)  BYTE      INITIAL (0).
      TOKEN       BYTE.      /*RETURNED FROM SCANNER */

      /* * * * * PROCEDURES USED BY THE SCANNER * * * */

185 1  NEXT$CHAR. PROCEDURE BYTE,
186 2  IF LOOKED THEN
187 2  DO,
188 3  LOOKED=FALSE,
189 3  RETURN (CHAR =HOLD);
190 3  END;
191 2  IF (POINTER:=POINTER + 1) >= BUFFER$END THEN
192 2  DO,
193 3  IF NOT MORE$INPUT THEN
194 3  DO,
195 4  BUFFER$END= MEMORY;
196 4  POINTER= ADD$END,
197 4  END;
198 3  ELSE POINTER=IN$BUFF;
199 3  END;
200 2  RETURN (CHAR:=NEXT);
201 2  END NEXT$CHAR;

202 1  GET$CHAR. PROCEDURE,
      /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS NEEDED WITHOUT
      THE DIRECT RETURN OF THE CHARACTER*/
203 2  CHAR=NEXT$CHAR;
204 2  END GET$CHAR;

205 1  DISPLAY$LINE. PROCEDURE,
206 2  IF NOT LIST$INPUT THEN RETURN,
207 2  DISPLAY(DISPLAY(0) + 1) = '$',
208 2  CALL PRINT( DISPLAY(1));
209 2  DISPLAY(0) = 0.
210 2  END DISPLAY$LINE;

211 2  END DISPLAY$LINE;

212 1  LOAD$DISPLAY. PROCEDURE,
213 2  IF DISPLAY(0) < 72 THEN
214 2  DISPLAY(DISPLAY(0) + 1) = CHAR,
215 2  CALL GET$CHAR;
216 2  END LOAD$DISPLAY;

217 1  PUT. PROCEDURE,
218 2  IF ACCUM(0) < ACCUM$LENG THEN
219 2  ACCUM(ACCUM(0) + 1) = CHAR,
220 2  CALL LOAD$DISPLAY;
221 2  END PUT;

222 1  EAT$LINE. PROCEDURE,
223 2  DO WHILE CHAR<>CR;
224 3  CALL LOAD$DISPLAY;
225 3  END;
226 2  END EAT$LINE;

227 1  GET$NO$BLANK. PROCEDURE,
228 2  DECLARE (N,I) BYTE,
229 2  DO FOREVER,
230 3  IF CHAR = ' ' THEN CALL LOAD$DISPLAY;
      ELSE
231 3  IF CHAR=CR THEN
232 3  DO,
233 4  CALL DISPLAY$LINE,
234 4  IF SEQ$NUM THEN N=3; ELSE N=2,
235 4  DO I = 1 TO N;
236 5  CALL LOAD$DISPLAY;
237 5  END;
238 4  IF CHAR = '*' THEN CALL EAT$LINE,
239 4  ELSE
240 4  IF CHAR = '.' THEN
241 5  DO,
242 6  IF NOT DEBUGGING THEN CALL EAT$LINE,
243 6  ELSE CALL LOAD$DISPLAY;
244 5  END;
245 4  END;
246 3  END;
247 3  END;
248 3  END;
      RETURN;
249 3  END; /* END OF DO FOREVER */
250 2  END GET$NO$BLANK;

251 1  SPACE. PROCEDURE BYTE,
252 2  RETURN (CHAR= ' ') OR (CHAR=CR);
253 2  END SPACE;

254 1  DELIMITER. PROCEDURE BYTE,
      /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR*/

```



```

157 2      IF CHAR <> ' ' THEN RETURN FALSE;
159 2      HOLD=NEXT$CHAR;
160 2      LOOKED=TRUE;
161 2      IF SPACE THEN
162 2      DO;
163 1          CHAR = ' ';
164 1          RETURN TRUE;
165 1      END;
166 2      CHAR=' ';
167 2      RETURN FALSE;
168 2  END DELIMITER;

169 1  END$OF$TOKEN  PROCEDURE BYTE;
170 2      RETURN SPACE OR DELIMITER;
171 2  END END$OF$TOKEN;

172 1  GET$LITERAL  PROCEDURE BYTE;
173 2      CALL LOAD$DISPLAY;
174 2      DO FOREVER;
175 3          IF CHAR= QUOTE THEN
176 3          DO;
177 4              CALL LOAD$DISPLAY;
178 4              RETURN LITERAL;
179 4          END;
180 3          CALL PUT;
181 3      END;
182 2  END GET$LITERAL;

183 1  LOOK$UP  PROCEDURE BYTE;
184 2      DECLARE POINT ADDRESS;
185 2      HERE BASED POINT (1) BYTE;
186 2      I          BYTE;
187 2
188 2      MATCH: PROCEDURE BYTE;
189 3          DECLARE J BYTE;
190 3          DO J=1 TO ACCUM(0);
191 3              IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
192 3          END;
193 3          RETURN TRUE;
194 2      END MATCH;

195 2      POINT=OFFSET(ACCUM(0))+ TABLE;
196 2      DO I=1 TO WORD$COUNT(ACCUM(0));
197 3          IF MATCH THEN RETURN I;
198 3          POINT = POINT + ACCUM(0);
199 3      END;
200 2      RETURN FALSE;
201 2  END LOOK$UP;

202 1  RESERVED$WORD  PROCEDURE BYTE;
203 2      /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE CONTENTS OF
204 2      THE ACCUMULATOR IS A RESERVED WORD, OTHERWISE RETURNS ZERO */
205 2      DECLARE VALUE BYTE;
206 2      DECLARE NUMB BYTE;
207 2      IF ACCUM(0) > MAX$LEN THEN RETURN 0;
208 2      IF (NUMB=TOKEN$TABLE(ACCUM(0)))=0 THEN RETURN 0;
209 2      IF (VALUE =LOOK$UP)=0 THEN RETURN 0;
210 2      RETURN (NUMB + VALUE);
211 2  END RESERVED$WORD;

212 1  GET$TOKEN  PROCEDURE BYTE;
213 2      ACCUM(0)=0;
214 2      CALL GET$NO$BLANK;
215 2      IF CHAR=QUOTE THEN RETURN GET$LITERAL;
216 2      IF DELIMITER THEN
217 2      DO;
218 3          CALL PUT;
219 3          RETURN PERIOD;
220 3      END;
221 2      DO FOREVER;
222 3          CALL PUT;
223 3          IF END$OF$TOKEN THEN RETURN INPUT$STR;
224 3      END; /* OF DO FOREVER */
225 2  END GET$TOKEN;

226 1  SCANNER  PROCEDURE;
227 2      DECLARE CHECK BYTE;
228 2      DO FOREVER;
229 3          IF (TOKEN =GET$TOKEN) = INPUT$STR THEN
230 3          IF (CHECK:=RESERVED$WORD) <> 0 THEN TOKEN=CHECK;
231 3          IF TOKEN <> 0 THEN RETURN;
232 3          CALL PRINT$ERROR ('SE');
233 3          DO WHILE NOT END$OF$TOKEN;
234 4              CALL GET$CHAR;

```



```

239 4      END.
240 1      END.
241 2      END SCANNER.

242 1      PRINT$ACCU. PROCEDURE.
243 2      ACCU<ACCU(0)+1>=$'.
244 2      CALL PRINT<ACCU(1)>.
245 2      END PRINT$ACCU.

246 1      PRINT$NUMBER. PROCEDURE<NUMB>.
247 2      DECLARE<NUMB,I,CNT,K> BYTE, J<*> BYTE DATA<100,10>.
248 2      DO I=0 TO 1.
249 3          CNT=0.
250 3          DO WHILE NUMB >= <K.=J<I>>.
251 4              NUMB=NUMB - K.
252 4              CNT=CNT + 1.
253 4          END.
254 3          CALL PRINTCHAR<'0' + CNT>.
255 3      END.
256 2      CALL PRINTCHAR<'0' + NUMB>.
257 2      END PRINT$NUMBER.

258 1      INIT$SCANNER. PROCEDURE.
259 2      DECLARE CON$CBL<*> BYTE DATA<'CBL'>.
260 2      /* INITIALIZE FOR INPUT - OUTPUT OPERATIONS */
261 2      CALL MOVE<CON$CBL, IN$ADDR + 5, 3>.
262 2      CALL FILL<IN$ADDR + 12,0,5>.
263 2      CALL OPEN.
264 2      CALL MOVE<IN$ADDR, OUTPUT$FCB,5>.
265 2      OUTPUT$FCB<2> = 0.
266 2      OUTPUT$END=<OUTPUT$PTR.= OUTPUT$BUFF - 1> + 128.
267 2      CALL MAKE.
268 2      CALL GET$CHAR. /* PRINE THE SCANNER */
269 2      DO WHILE CHAR = '$'.
270 3          IF NEXTCHAR = 'L' THEN LIST$INPUT=NOT LIST$INPUT.
271 3          ELSE IF CHAR = 'S' THEN SEQ$NUM= NOT SEQ$NUM.
272 3          ELSE IF CHAR = 'P' THEN PRINT$PROD = NOT PRINT$PROD.
273 3          ELSE IF CHAR = 'T' THEN PRINT$TOKEN = NOT PRINT$TOKEN.
274 3          CALL GET$CHAR.
275 3          CALL GET$NO$BLANK.
276 3      END.
277 2      END INIT$SCANNER.

/* * * * * END OF SCANNER PROCEDURES * * * */

/* * * * * SYMBOL TABLE DECLARATIONS * * * */

281 1      DECLARE
CUR$SYM          ADDRESS.      /*SYMBOL BEING ACCESSED*/
SYMBOL           BASED CUR$SYM<1> BYTE.
SYMBOL$ADDR      BASED CUR$SYM<1> ADDRESS.
NEXT$SYM$ENTRY   BASED NEXT$SYM ADDRESS.
HASH$PTR         ADDRESS.
DISPLACEMENT     LIT          '12'.
HASH$MASK        LIT          '3FH'.
S$TYPE           LIT          '2'.
OCCURS           LIT          '11'.
ADDR2            LIT          '4'.
P$LENGTH         LIT          '3'.
S$LENGTH         LIT          '3'.
LEVEL            LIT          '10'.
LOCATION           LIT          '2'.
REL$ID           LIT          '5'.
START$NAME       LIT          '11'. /*1 LESS*/
MAX$ID$LEN       LIT          '12'.

/* * * * * TYPE LITERALS * * * * */

282 1      DECLARE
SEQUENTIAL       LIT          '1'.
RANDOM            LIT          '2'.
SEQ$RELATIVE     LIT          '3'.
VARIABLE$LENG    LIT          '4'.
GROUP            LIT          '5'.
COMP             LIT          '21'.

/* * * * * SYMBOL TABLE ROUTINES * * * */

283 1      INIT$SYMBOL. PROCEDURE.
284 2      CALL FILL<FREE$STORAGE,0,130>.
285 2      /* INITIALIZE HASH TABLE AND FIRST COLLISION FIELD */
286 2      NEXT$SYM=FREE$STORAGE+128.

```



```

HOLD#SYM          ADDRESS.
PENDING#LITERAL  BYTE INITIAL(FALSE).
PENDING#LIT#ID   ADDRESS.
REDEF            BYTE    INITIAL (FALSE).
REDEF#ONE        ADDRESS.
REDEF#TWO        ADDRESS.
TEMP#HOLD        ADDRESS.
TEMP#TWO         ADDRESS.
COMPILING        BYTE    INITIAL(TRUE).
SP              BYTE    INITIAL (255).
MP              BYTE.
MPP1            BYTE.
NOLOOK          BYTE    INITIAL(TRUE).
(I, J, K)        BYTE    /*INDICIES FOR THE PARSE*/
STATE           BYTE    INITIAL(STARTS).

/* * * * * PARSE ROUTINES * * * * */

345 1  BYTE$OUT  PROCEDURE(ONE#BYTE);
      /* THIS PROCEDURE WRITES ONE BYTE OF OUTPUT ONTO THE DISK
346 2  IF REQUIRED THE OUTPUT BUFFER IS DUMPED TO THE DISK */
347 2  DECLARE ONE#BYTE BYTE.
348 2  IF (OUTPUT#PTR = OUTPUT#PTR + 1) OUTPUT#END THEN
349 2  DO;
350 3  CALL WRITE$OUTPUT;
351 3  OUTPUT#PTR = OUTPUT#BUFF;
352 2  END;
353 2  OUTPUT#CHAR = ONE#BYTE.
      END BYTE$OUT.

354 1  STRING$OUT PROCEDURE (ADDR, COUNT);
355 2  DECLARE (ADDR, I, COUNT) ADDRESS. (CHAR BASED ADDR) BYTE.
356 2  DO I=1 TO COUNT.
357 3  CALL BYTE$OUT(CHAR).
358 3  ADDR=ADDR+1.
359 2  END;
360 2  END STRING$OUT.

361 1  ADDR$OUT  PROCEDURE(ADDR);
362 2  DECLARE ADDR ADDRESS.
363 2  CALL BYTE$OUT(LOW(ADDR));
364 2  CALL BYTE$OUT(HIGH(ADDR));
365 2  END ADDR$OUT.

366 1  FILL$STRING PROCEDURE(COUNT, CHAR);
367 2  DECLARE (I, COUNT) ADDRESS. CHAR BYTE.
368 2  DO I=1 TO COUNT;
369 3  CALL BYTE$OUT(CHAR);
370 2  END;
371 2  END FILL$STRING.

372 1  START$INITIALIZE PROCEDURE(ADDR, CNT);
373 2  DECLARE (ADDR, CNT) ADDRESS.
374 2  CALL BYTE$OUT(INT);
375 2  CALL ADDR$OUT(ADDR);
376 2  CALL ADDR$OUT(CNT);
377 2  END START$INITIALIZE.

378 1  BUILD$SYMBOL PROCEDURE(LEN);
379 2  DECLARE LEN BYTE, TEMP ADDRESS.
380 2  TEMP#NEXT#SYM.
381 2  IF (NEXT#SYM = SYMBOL(LEN, LEN-DISPLACEMENT))
382 3  > MAXMEMORY THEN CALL FATAL$ERROR('ST');
383 2  CALL FILL (TEMP, 0, LEN).
384 2  END BUILD$SYMBOL.

385 1  MATCH. PROCEDURE ADDRESS.
      /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
      TABLE. IF IT IS PRESENT, CUR#SYM IS SET FOR ACCESS
      OTHERWISE A NEW ENTRY IS MADE AND THE PRINT NAME
      IS ENTERED. ALL NAMES ARE TRUNCATED TO MAX#ID#LEN*/
386 2  DECLARE POINT ADDRESS.
      COLLISION BASED POINT ADDRESS.
      (HOLD, I) BYTE;
387 2  IF VARC(0)>MAX#ID#LEN
      THEN VARC(0) = MAX#ID#LEN.
      /* TRUNCATE IF REQUIRED */
388 2  HOLD = 0;
389 2  DO I=1 TO VARC(0); /* CALCULATE HASH CODE */
390 3  HOLD=HOLD + VARC(I);
391 2  END;
392 2  POINT=FREE$STORAGE + SHL((HOLD AND HASH$MASK), 1);
393 2  DO FOREVER;
394 3  IF COLLISION=0 THEN
395 4  DO;
396 5  CUR#SYM, COLLISION=NEXT#SYM;
397 5  CALL BUILD$SYMBOL(VARC(0));
398 4

```



```

399 4          /* LOAD PRINT NAME */
400 4          SYMBOL($LENGTH)=VARC(0),
401 4          DO I = 1 TO VARC(0),
402 4              SYMBOL(START$NAME + I)=VARC(I),
403 4          END,
404 4          RETURN CUR$SYM,
          END,
          ELSE
          DO,
405 3              CUR$SYM=COLLISION,
406 4              IF (HOLD =GET$P$LENGTH)=VARC(0) THEN
407 4                  DO,
408 4                      I=1,
409 4                      DO WHILE SYMBOL(START$NAME + I)= VARC(I),
410 4                          IF (I =I+1)>HOLD THEN RETURN (CUR$SYM =COLLISION),
411 4                          IF (I =I+1)>HOLD THEN RETURN (CUR$SYM =COLLISION),
412 4                          END,
413 4                      END,
414 4                  END,
415 4                  POINT=COLLISION,
416 4              END,
417 4              END MATCH,
418 2
419 1  ALLOCATE, PROCEDURE(BYTES$REQ) ADDRESS,
          /* THIS ROUTINE CONTROLS THE ALLOCATION OF SPACE
          IN THE MEMORY OF THE INTERPRETER */
420 2  DECLARE (HOLD,BYTES$REQ) ADDRESS,
421 2  HOLD=NEXT$AVAILABLE,
422 2  IF (NEXT$AVAILABLE =NEXT$AVAILABLE + BYTES$REQ)>MAX$INT$MEM
          THEN CALL FATAL$ERROR('MO'),
424 2  RETURN HOLD,
425 2  END ALLOCATE,
426 1  SET$REDEF PROCEDURE(OLD,NEW),
427 2  DECLARE (OLD,NEW) ADDRESS,
428 2  IF (REDEF =NDT REDEF) THEN
429 2      DO,
430 3          REDEF$ONE=OLD,
431 3          REDEF$TWO=NEW,
432 3      END,
433 2  ELSE CALL PRINT$ERRDR('R1'),
434 2  END SET$REDEF,
435 1  SET$CUR$SYM, PROCEDURE,
436 2  CUR$SYM=ID$STACK(ID$STACK$PTR),
437 2  END SET$CUR$SYM,
438 1  STACK$LEVEL, PROCEDURE BYTE,
439 2  CALL SET$CUR$SYM,
440 2  RETURN GET$LEVEL,
441 2  END STACK$LEVEL,
442 1  LOAD$LEVEL, PROCEDURE,
443 2  DECLARE HOLD ADDRESS,
444 2  LOAD$REDEF$ADDR, PROCEDURE,
445 3  CUR$SYM=REDEF$ONE,
446 3  HOLD=GET$ADDRESS,
447 3  END LOAD$REDEF$ADDR,
448 2  IF ID$STACK(0) < 0 THEN
449 2      DO,
450 3          IF VALUE(SP-2)=0 THEN
451 3              DO,
452 4                  CALL SET$CUR$SYM,
453 4                  HOLD=GET$P$LENGTH + GET$ADDRESS,
454 4              END,
455 3          ELSE CALL LOAD$REDEF$ADDR,
456 3          IF (ID$STACK$PTR =ID$STACK$PTR+1)>9 THEN
457 3              DO,
458 4                  CALL PRINT$ERRDR('EL '),
459 4                  ID$STACK$PTR=9,
460 4              END,
461 3          END,
462 2          ELSE HOLD=NEXT$AVAILABLE,
463 2          ID$STACK(ID$STACK$PTR)=VALUE(MPP1),
464 2          CALL SET$CUR$SYM,
465 2          CALL SET$ADDRESS(HOLD),
466 2  END LOAD$LEVEL,
467 1  REDEF$DR$VALUE, PROCEDURE,
468 2  DECLARE HOLD ADDRESS,
          (DEC, N, J, SIGN) BYTE,
469 2  IF REDEF THEN
470 2      DO,
471 3          IF REDEF$TWO=CUR$SYM THEN
472 3              DO,

```



```

473 4          HOLD=GET$S$LENGTH;
474 4          CUR$SYM=REDEF$ONE;
475 4          IF HOLD<GET$S$LENGTH THEN
476 4          DO;
477 5              CALL PRINT$ERROR('R2');
478 5              HOLD=GET$S$LENGTH;
479 5              CUR$SYM=REDEF$ONE;
480 5              CALL SET$S$LENGTH(HOLD);
481 5          END;
482 4          REDEF=FALSE;
483 4      END;
484 3  END;
485 2  ELSE IF PENDING$LITERAL=0 THEN RETURN;
      IF PENDING$LIT$ID<ID$STACK$PTR THEN RETURN;
489 2  CALL START$INITIALIZE(GET$ADDRESS, HOLD =GET$S$LENGTH);
490 2  IF PENDING$LITERAL>2 THEN
491 2  DO;
492 3      IF PENDING$LITERAL=3 THEN CHAR='0';
493 3      ELSE IF PENDING$LITERAL=4 THEN CHAR=' ' ;
494 3      ELSE CHAR=QUOTE;
495 3      CALL FILL$STRING(HOLD, CHAR);
496 3  END;
497 2  ELSE IF PENDING$LITERAL = 2 THEN
498 2  DO;
499 3      IF HOLD <= HOLD$LIT(0) THEN
500 3          CALL STRING$OUT( HOLD$LIT(1),HOLD);
501 3      ELSE DO;
502 4          CALL STRING$OUT( HOLD$LIT(1),HOLD$LIT(0));
503 4          CALL FILL$STRING(HOLD - (HOLD$LIT(0) + 1), ' ');
504 4      END;
505 2  END;
506 2  ELSE DO;
507 3      /* THE NUMBER HANDLER */
508 3      DECLARE (DEC,MINUS$SIGN,I,J,LIT$DEC,N$LENGTH,
509 3          NUM$BEFORE,NUM$AFTER, TYPE) BYTE, ZONE LIT '10H';
510 3      IF (TYPE =GET$TYPE)<16) OR (TYPE>20) THEN
511 3          CALL PRINT$ERROR('NV');
512 3      N$LENGTH=GET$S$LENGTH;
513 3      DEC=GET$DECIMAL;
514 3      MINUS$SIGN=FALSE;
515 3      IF HOLD$LIT(1) = '-' THEN
516 3      DO;
517 4          MINUS$SIGN=TRUE;
518 4          J=1;
519 4      END;
520 3      ELSE IF HOLD$LIT(1) = '+' THEN J=1;
521 3      ELSE J=0;
522 3      LIT$DEC=0;
523 3      DO I=1 TO HOLD$LIT(0);
524 4          IF HOLD$LIT(I)='.' THEN LIT$DEC=I;
525 4      END;
526 3      IF LIT$DEC=0 THEN
527 3      DO;
528 4          NUM$BEFORE=HOLD$LIT(1)-J;
529 4          NUM$AFTER=0;
530 4      END;
531 3      ELSE DO;
532 4          NUM$BEFORE=LIT$DEC -J-1;
533 4          NUM$AFTER=HOLD$LIT(1) - LIT$DEC;
534 4      END;
535 3      IF (I =N$LENGTH - DEC)<NUM$BEFORE THEN
536 3          CALL PRINT$ERROR('SL');
537 3      IF I>NUM$BEFORE THEN
538 3      DO;
539 4          I=I-NUM$BEFORE;
540 4          IF MINUS$SIGN THEN
541 5              I=I-1;
542 5          CALL BYTE$OUT('0' + ZONE);
543 5      END;
544 3      CALL FILL$STRING(I, '0');
545 3      ELSE IF MINUS$SIGN THEN HOLD$LIT(J-1)=HOLD$LIT(J-1)-ZONE;
546 3      CALL STRING$OUT( HOLD$LIT(1) + J, NUM$BEFORE);
547 3      IF NUM$AFTER > DEC THEN NUM$AFTER = DEC;
548 3      CALL STRING$OUT( HOLD$LIT(1) + LIT$DEC, NUM$AFTER);
549 3      IF (I =DEC - NUM$AFTER)<>0 THEN
550 3          CALL FILL$STRING(I, '0');
551 3  END;
552 2  PENDING$LITERAL=0;
553 2  END REDEF$OR$VALUE;
554 2
555 1  REDUCE$STACK PROCEDURE;
556 1  DECLARE HOLD$LENGTH ADDRESS;
557 1  CALL SET$CUR$SYM;
558 1  CALL REDEF$OR$VALUE

```



```

564 1      HOLD$LENGTH=GET$S$LENGTH.
565 2      IF GET$TYPE > 128 THEN
566 3      DO,
567 4      HOLD$LENGTH=HOLD$LENGTH + GET$OCCURS,
568 5      END,
569 6      ID$STACK$PTR=ID$STACK$PTR - 1,
570 7      CALL SET$CUR$SYN,
571 8      CALL SET$S$LENGTH<GET$S$LENGTH + HOLD$LENGTH>,
572 9      CALL SET$TYPE<GROUP>,
573 10     END REDUCE$STACK,

574 1      END$OP$RECORD: PROCEDURE,
575 2      DO WHILE ID$STACK$PTR<0,
576 3      CALL REDUCE$STACK,
577 4      END,
578 5      CALL SET$CUR$SYN,
579 6      CALL REDEF$OP$VALUE,
580 7      ID$STACK(0)=0,
581 8      TEMP$HOLD=ALLOCATE TEMP$TWO *GET$S$LENGTH>,
582 9      END END$OP$RECORD,

583 1      CONVERT$INTEGER: PROCEDURE,
584 2      DECLARE INTEGER ADDRESS,
585 3      INTEGER=0,
586 4      DO I = 1 TO VARC(0),
587 5      INTEGER=SHL(INTEGER,3)+SHL(INTEGER,1)+<VARC(I)-0>,
588 6      END,
589 7      VALUE$P/=INTEGER,
590 8      END CONVERT$INTEGER,

591 1      OR$VALUE: PROCEDURE(PTR,ATTRIB),
592 2      DECLARE PTR BYTE, ATTRIB ADDRESS,
593 3      VALUE$PTR)=VALUE$PTR OR ATTRIB,
594 4      END OR$VALUE,

595 1      BUILD$FCB: PROCEDURE,
596 2      DECLARE TEMP ADDRESS,
597 3      DECLARE BUFFER(11) BYTE, <CHAR, 1, 1> BYTE,
598 4      CALL FILL<BUFFER, 1, 11>,
599 5      J=0,
600 6      DO WHILE <J < 11> AND <IC VARC(0)>,
601 7      IF <CHAR =VARC(I) =1>>=1 THEN J=8,
602 8      ELSE DO,
603 9      BUFFER(J)=CHAR,
604 10     J=J+1,
605 11     END,
606 12     END,
607 13     CALL SET$ADDR(TEMP =ALLOCATE(164)),
608 14     CALL START$INITIALIZE(TEMP,16),
609 15     CALL BYTE$OUT(0),
610 16     CALL STRING$OUT< BUFFER, 11>,
611 17     CALL FILL$STRING(4,0),
612 18     CALL OR$VALUE$P=1,1>,
613 19     END BUILD$FCB,

614 1      SET$SIGN: PROCEDURE(NUMB),
615 2      DECLARE NUMB BYTE,
616 3      IF GET$TYPE=17 THEN CALL SET$TYPE<VALUE$P> + NUMB>,
617 4      ELSE CALL PRINT$ERR<OP 13>,
618 5      IF VALUE$P<0> THEN CALL SET$S$LENGTH<GET$S$LENGTH + 1>,
619 6      END SET$SIGN,

620 1      PIC$ANALYZER: PROCEDURE,
621 2      DECLARE /* WORK AREAS AND VARIABLES */
622 3      FLAG      BYTE,
623 4      FIRST     BYTE,
624 5      COUNT     ADDRESS,
625 6      BUFFER (31) BYTE,
626 7      SAVE      BYTE,
627 8      REPITITIONS ADDRESS,
628 9      J         BYTE,
629 10     DEC$COUNT BYTE,
630 11     CHAR      BYTE,
631 12     I         BYTE,
632 13     TEMP      ADDRESS,
633 14     TYPE      BYTE,

634 15     /* * * MASKS * * */
635 16     ALPHA    LIT '0',
636 17     A$EDIT   LIT '2',
637 18     A$N      LIT '4',
638 19     E$DIT    LIT '8',
639 20     NUM      LIT '16',
640 21     NUM$EDIT LIT '32',
641 22     DEC      LIT '64',
642 23     SIGN     LIT '128',

```



```

NUMSMASK      LIT      10101111B',
NUMSED$MASK   LIT      10000101B',
$NUM$MASK     LIT      '00101111B',
$SE$MASK      LIT      11111100B',
$EN$MASK      LIT      11101010B',
$SNSE$MASK    LIT      '11100000B'.

/* TYPES */
NETYPE LIT '80',
NTYPE  LIT '15',
SNTYPE LIT '17',
ATYPE  LIT '8',
RETYPE LIT '72',
ANTYPE LIT '3',
ANETYPE LIT '73',

625 2      INC$COUNT PROCEDURE(SWITCH),
626 1          DECLARE SWITCH BYTE,
627 0          FLAG=FLAG OR SWITCH,
628 1          IF <COUNT =COUNT + 1> < 31 THEN BUFFER(COUNT) = CHAR,
629 0      END INC$COUNT,

631 2      CHECK PROCEDURE(MASK) BYTE,
        /* THIS ROUTINE CHECKS A MASK AGAINST THE
        FLAG BYTE AND RETURNS TRUE IF THE FLAG
        HAD NO BITS IN COMMON WITH THE MASK */
632 0          DECLARE MASK BYTE,
633 1          RETURN NOT <(FLAG AND MASK) < 0>,
634 0      END CHECK,

635 2      PIC$ALLOCATE PROCEDURE(AMT) ADDRESS,
636 0          DECLARE AMT ADDRESS,
637 1          IF (MAX$INT$MEM = MAX$INT$MEM - AMT) < NEXT$AVAILABLE
        THEN CALL FATAL$ERROR('NO'),
638 0          RETURN MAX$INT$MEM,
639 1      END PIC$ALLOCATE,

/* PROCEDURE EXECUTION STARTS HERE */

641 2      COUNT, FLAG, DEC$COUNT=0,
        /* CHECK FOR EXCESSIVE LENGTH */
642 2      IF VARC(0) > 30 THEN
643 2          DO,
644 0              CALL PRINT$ERROR('PC'),
645 0              RETURN,
646 0      END,
        /* SET FLAG BITS AND COUNT LENGTH */
647 2      I =1,
648 0      DO WHILE (I=VARC(0))
649 0          IF (CHAR=VARC(I))='A' THEN CALL INC$COUNT(ALPHA),
650 0          ELSE IF CHAR='B' THEN CALL INC$COUNT(AR$EDIT),
651 0          ELSE IF CHAR='9' THEN CALL INC$COUNT(NUM),
652 0          ELSE IF CHAR='X' THEN CALL INC$COUNT(RN),
653 0          ELSE IF (CHAR='S') AND (COUNT=0) THEN
654 0              FLAG=FLAG OR SIGN,
655 0          ELSE IF (CHAR='V') AND (DEC$COUNT=0) THEN
656 0              DEC$COUNT=COUNT,
657 0          ELSE IF (CHAR='/' OR (CHAR='0') THEN CALL INC$COUNT(EDIT),
658 0          ELSE IF
        (CHAR='C') OR (CHAR='P') OR (CHAR='*') OR
        (CHAR='<') OR (CHAR='>') OR (CHAR='$') THEN
659 0              CALL INC$COUNT(NUM$EDIT),
660 0          ELSE IF (CHAR=' ') AND (DEC$COUNT=0) THEN
661 0              DO,
662 0                  CALL INC$COUNT(NUM$EDIT),
663 0                  DEC$COUNT=COUNT,
664 0              END,
665 0          ELSE IF ((CHAR='C') AND (VARC(I+1)='R')) OR
        ((CHAR='D') AND (VARC(I+1)='B')) THEN
666 0              DO,
667 0                  CALL INC$COUNT(NUM$EDIT),
668 0                  CHAR=VARC(I+1),
669 0                  CALL INC$COUNT(NUM$EDIT),
670 0              END,
671 0          ELSE IF (CHAR='<') AND (COUNT=0) THEN
672 0              DO,
673 0                  SAVE=VARC(I+1),
674 0                  REPETITIONS=0,
675 0                  DO WHILE(CHAR =VARC(I+1))<D>
        REPETITIONS=SHL(REPETITIONS, 2) +
        SHL(REPETITIONS, 1) +(CHAR -'0'),
676 0                  END,
677 0                  CHAR=SAVE,
678 0                  DO J=1 TO REPETITIONS-1,
679 0                      CALL INC$COUNT(0),
680 0                  END,
681 0              END,
682 0          END,
683 0      END,
684 0      END,
685 0      END,
686 0      END,
687 0      END,

```



```

688 3      ELSE DO,
689 4          CALL PRINT$ERROR('PC'),
690 4          RETURN,
691 4      END,
692 3      I=I+1,
693 3      END, /* END OF DO WHILE IC= VARC */
/* AT THIS POINT THE TYPE CAN BE DETERMINED */
IF NOT CHECK(NUM$EDIT) THEN
694 2      DO,
695 2          IF CHECK(NUM$MASK) THEN TYPE=NETYPE,
696 3      END,
697 2      ELSE IF CHECK(NUM$MASK) THEN TYPE=ATYPE,
701 2      ELSE IF CHECK(SNUM$MASK) THEN TYPE=SSN$TYPE,
703 2      ELSE IF CHECK(NDT(ALPHA)) THEN TYPE=ATYPE,
705 2      ELSE IF CHECK(CASE$MASK) THEN TYPE=ATYPE,
707 2      ELSE IF CHECK(CASE$MASK) THEN TYPE=ATYPE,
709 2      ELSE IF CHECK(CASE$MASK) THEN TYPE=ATYPE,
IF TYPE=0 THEN CALL PRINT$ERROR('PC'),
ELSE DO,
IF REDEF THEN CUR$SYM=REDEF$TWD,
ELSE CUR$SYM = HOLD$SYM,
CALL SET$TYPE(TYPE),
CALL SET$LENGTH(COUNT + GET$S$LENGTH),
IF (TYPE AND 64) <> 0 THEN
720 4      DO,
721 4          CALL SET$ADDR2(TEMP, PIC$ALLOCATE(COUNT)),
722 4          CALL START$INITIALIZE(TEMP, COUNT),
723 4          CALL STRING$OUT( BUFFER + 1, COUNT),
724 4      END,
725 3      IF DEC$COUNT<0 THEN CALL SET$DECIMAL(COUNT-DEC$COUNT),
726 2      END,
END PIC$ANALYZER,

729 1      SET$FILE$ATTRIB PROCEDURE,
730 2      DECLARE TEMP ADDRESS, TYPE BYTE,
731 2      IF CUR$SYM<>VALUE(MPP1) THEN
732 2      DO,
733 3          TEMP=CUR$SYM,
734 3          CUR$SYM=VALUE(MPP1),
735 3          SYMBOL$ADDR(REL$ID)=TEMP,
736 3      END,
737 2      IF NOT (TEMP=VALUE(SP-1)) THEN CALL PRINT$ERROR('NFP'),
738 2      ELSE DO,
739 3          IF TEMP=1 THEN TYPE=SEQUENTIAL,
740 3          ELSE IF TEMP=15 THEN TYPE=RANDOM,
741 3          ELSE IF TEMP=9 THEN TYPE=SEQ$RELATIVE,
742 3          ELSE DO,
743 4              CALL PRINT$ERROR('IA'),
744 4              TYPE=1,
745 4          END,
746 3      END,
747 2      CALL SET$TYPE(TYPE),
748 2      END SET$FILE$ATTRIB,

753 1      LOAD$LITERAL PROCEDURE,
754 2      DECLARE I BYTE,
755 2      IF PENDING$LITERAL <> 0 THEN CALL PRINT$ERROR('LE'),
756 2      ELSE DO I = 0 TO VARC(0),
757 3          HOLD$LIT(I)=VARC(I),
758 3      END,
759 2      END LOAD$LITERAL,

761 1      CHECK$FOR$LEVEL PROCEDURE,
762 2      DECLARE NEW$LEVEL BYTE,
763 2      HOLD$SYM, CUR$SYM=VALUE(MP-1),
764 2      CALL SET$LEVEL(NEW$LEVEL:=VALUE(MP-2)),
765 2      IF NEW$LEVEL=1 THEN
766 2      DO,
767 3          IF ID$STACK(0)<0 THEN
768 3          DO,
769 4              IF NOT FILE$SEC$END THEN
770 4              DO,
771 5                  CALL SET$REDEF(ID$STACK(0), VALUE(MP-1)),
772 5                  VALUE(MP)=1, /* SET REDEFINE FLAG */
773 5              END,
774 4              CALL END$OF$RECORD,
775 4          END,
776 3          END,
777 2      ELSE DO WHILE STACK$LEVEL >= NEW$LEVEL,
778 3          CALL REDUCE$STACK,
779 3      END,
780 2      END CHECK$FOR$LEVEL,

781 1      CODE$GEN PROCEDURE( PRODUCTION),
782 2      DECLARE PRODUCTION BYTE,

```



```

783 2      IF PRINT$PROD THEN
784 2      DO,
785 3          CALL CALF,
786 3          CALL PRINTCHAR(FOUND),
787 3          CALL PRINT$NUMBER(PRODUCTION);
788 2      END;

789 2      DO CASE PRODUCTION;

/* P R O D U C T I O N S */

/* CASE 0 NOT USED */

790 3      /*
791 3      /* 1 <PROGRAM> = <ID-DIV> <E-DIV> <D-DIV> PROCEDURE
792 3      COMPILING=FALSE,
793 3      /* 2 <ID-DIV> = IDENTIFICATION DIVISION PROGRAM-ID
794 3      /* 2 <COMMENT> <AUTH> <DATE> <SEC>
795 3      /* 3 /* NO ACTION REQUIRED */
796 3      /* 3 <AUTH> = AUTHOR <COMMENT>
797 3      /* 4 /* NO ACTION REQUIRED */
798 3      /* 4 \! <EMPTY>
799 3      /* 5 /* NO ACTION REQUIRED */
800 3      /* 5 <DATE> = DATE-WRITTEN <COMMENT>
801 3      /* 6 /* NO ACTION REQUIRED */
802 3      /* 6 \! <EMPTY>
803 3      /* 7 /* NO ACTION REQUIRED */
804 3      /* 7 <SEC> = SECURITY <COMMENT>
805 3      /* 8 /* NO ACTION REQUIRED */
806 3      /* 8 \! <EMPTY>
807 3      /* 9 /* NO ACTION REQUIRED */
808 3      /* 9 <COMMENT> = <INPUT>
809 3      /* 10 /* NO ACTION REQUIRED */
810 3      /* 10 \! <COMMENT> <INPUT>
811 3      /* 11 /* NO ACTION REQUIRED */
812 3      /* 11 <E-DIV> = ENVIRONMENT DIVISION CONFIGURATION
813 3      /* 11 SECTION <SRC-OBJ> <I-O>
814 3      /* 12 /* NO ACTION REQUIRED */
815 3      /* 12 <SRC-OBJ> = SOURCE-COMPUTER <COMMENT> <DEBUG>
816 3      /* 12 OBJECT-COMPUTER <COMMENT>
817 3      /* 13 /* NO ACTION REQUIRED */
818 3      /* 13 <DEBUG> = DEBUGGING MODE
819 3      DEBUGGING=TRUE, /* SETS A SCANNER TOGGLE */
820 3      /* 14 /* NO ACTION REQUIRED */
821 3      /* 14 \! <EMPTY>
822 3      /* 15 /* NO ACTION REQUIRED */
823 3      /* 15 <I-O> = INPUT-OUTPUT SECTION FILE-CONTROL
824 3      /* 15 <FILE-CONTROL-LIST> <IC>
825 3      /* 16 /* NO ACTION REQUIRED */
826 3      /* 16 \! <EMPTY>
827 3      /* 17 /* NO ACTION REQUIRED */
828 3      /* 17 <FILE-CONTROL-LIST> = <FILE-CONTROL-ENTRY>
829 3      /* 18 /* NO ACTION REQUIRED */
830 3      /* 18 \! <FILE-CONTROL-LIST>
831 3      /* 18 <FILE-CONTROL-ENTRY>
832 3      /* 19 /* NO ACTION REQUIRED */
833 3      /* 19 <FILE-CONTROL-ENTRY> = SELECT <ID> <ATTRIBUTE-LIST>
834 3      CALL SET$FILE$ATTRIB;
835 3      /* 20 <ATTRIBUTE-LIST> = <ONE-ATTRIB>
836 3      /* 21 /* NO ACTION REQUIRED */
837 3      /* 21 \! <ATTRIBUTE-LIST> <ONE-ATTRIB>
838 3      /* 22 VALUE(MP)=VALUE(SP) OR VALUE(MP);
839 3      /* 22 <ONE-ATTRIB> = ORGANIZATION <ORG-TYPE>
840 3      /* 23 VALUE(MP)=VALUE(SP);
841 3      /* 23 \! ACCESS <ACC-TYPE> <RELATIVE>
842 3      /* 24 VALUE(MP)=VALUE(MPP1) OR VALUE(SP);
843 3      /* 24 \! ASSIGN <INPUT>
844 3      CALL BUILD$PCB;
845 3      /* 25 <ORG-TYPE> = SEQUENTIAL
846 3      /* 26 /* NO ACTION REQUIRED - DEFAULT */
847 3      /* 26 \! RELATIVE
848 3      /* 27 CALL OR$VALUE(SP,4);
849 3      /* 27 <ACC-TYPE> = SEQUENTIAL
850 3      /* 28 /* NO ACTION REQUIRED - DEFAULT */
851 3      /* 28 \! RANDOM
852 3      /* 29 CALL OR$VALUE(SP,2);
853 3      /* 29 <RELATIVE> = RELATIVE <IO>
854 3      /* 30 CALL OR$VALUE(MP,3);
855 3      /* 30 \! <EMPTY>
856 3      /* 31 /* NO ACTION REQUIRED - DEFAULT */
857 3      /* 31 <IC> = I-O-CONTROL <SAME-LIST>
858 3      /* 32 \! <EMPTY>
859 3      /* 33 <SAME-LIST> = <SAME-ELEMENT>
860 3      /* 34 \! <SAME-LIST> <SAME-ELEMENT>
861 3      /* 35 <SAME-ELEMENT> = SAME <ID-STRING>

```



```

823 3      /* 16 <ID-STRING> = <ID> */
826 3      /* 17      \! <ID-STRING> <ID> */
827 3
      /* 18 <D-DIV> = DATA DIVISION <FILE-SECTION> <WORK> */
      /* 19      <LINK> */
828 3      /* 20 NO ACTION REQUIRED */
      /* 21 <FILE-SECTION> = FILE SECTION <FILE-LIST> */
829 3      FILE$SEC$END = TRUE;
      /* 22      \! <EMPTY> */
830 3      FILE$SEC$END=TRUE;
      /* 23 <FILE-LIST> = <FILES> */
831 3      /* 24 NO ACTION REQUIRED */
      /* 25      \! <FILE-LIST> <FILES> */
832 3      /* 26 NO ACTION REQUIRED */
      /* 27 <FILES> = FD <ID> <FILE-CONTROL> */
      /* 28      <RECORD-DESCRIPTION> */
      DO,
833 3          CALL END$OF$RECORD;
834 4          CUR$SYM=VALUE(MP1);
835 4          CALL SET$ADDRESS(TEMP$HOLD);
836 4          CALL SET$LENGTH(TEMP$TWO);
837 4      END;
      /* 29 <FILE-CONTROL> = <FILE-LIST> */
838 3      /* 30 NO ACTION REQUIRED */
      /* 31      \! <EMPTY> */
840 3      /* 32 NO ACTION REQUIRED */
      /* 33 <FILE-LIST> = <FILE-ELEMENT> */
841 3      /* 34 NO ACTION REQUIRED */
      /* 35      \! <FILE-LIST> <FILE-ELEMENT> */
842 3      /* 36 NO ACTION REQUIRED */
      /* 37 <FILE-ELEMENT> = BLOCK <INTEGER> RECORDS */
843 3      /* 38 NO ACTION REQUIRED - FILES NEVER BLOCKED */
      /* 39      \! RECORD <REC-COUNT> */
844 3      CALL SET$LENGTH(VALUE$P);
      /* 40      \! LABEL RECORDS STANDARD */
845 3      /* 41 NO ACTION REQUIRED */
      /* 42      \! LABEL RECORDS OMITTED */
846 3      /* 43 NO ACTION REQUIRED */
      /* 44      \! VALUE OF <ID-STRING> */
847 3      /* 45 NO ACTION REQUIRED */
      /* 46 <REC-COUNT> = <INTEGER> */
848 3      /* 47 NO ACTION REQUIRED - VALUE$P CORRECT */
      /* 48      \! <INTEGER> TO <INTEGER> */
      DO,
849 3          VALUE(MP)=VALUE$P; /* VARIABLE LENGTH */
850 4          CALL SET$TYPE(4); /* SET TO VARIABLE */
851 4      END;
      /* 49 <WORK> = WORKING-STORAGE SECTION */
      /* 50      <RECORD-DESCRIPTION> */
853 3      /* 51 NO ACTION REQUIRED */
      /* 52      \! <EMPTY> */
854 3      /* 53 NO ACTION REQUIRED */
      /* 54 <LINK> = LINKAGE SECTION <RECORD-DESCRIPTION> */
855 3      CALL PRINT$ERROR('N1'); /* INTER PROG COMM */
      /* 55      \! <EMPTY> */
856 3      /* 56 NO ACTION REQUIRED */
      /* 57 <RECORD-DESCRIPTION> = <LEVEL-ENTRY> */
857 3      /* 58 NO ACTION REQUIRED */
      /* 59      \! <RECORD-DESCRIPTION> <LEVEL-ENTRY> */
858 3      /* 60 NO ACTION REQUIRED */
      /* 61 <LEVEL-ENTRY> = <INTEGER> <DATA-ID> <REDEFINES> */
      /* 62      <DATA-TYPE> */
      DO,
859 3          CALL LOAD$LEVEL;
860 4          IF PENDING$LITERAL(0) THEN PENDING$LIT$ID=ID$STACK$PTR;
861 4      END;
      /* 63 <DATA-ID> = <ID> */
864 3      /* 64 NO ACTION REQUIRED */
      /* 65      \! FILLER */
      DO,
866 4          CUR$SYM=VALUE$P;NEXT$SYM;
867 4          CALL BUILD$SYMBOL(0);
868 4      END;
      /* 66 <REDEFINES> = REDEFINES <ID> */
      DO,
869 3          CALL SET$REDEF(VALUE$P,VALUE$P-2);
870 4          VALUE(MP)=1; /* SET REDEFINE FLAG ON */
871 4          CALL CHECK$FOR$LEVEL;
872 4      END;
      /* 67      \! <EMPTY> */
874 3      CALL CHECK$FOR$LEVEL;
      /* 68 <DATA-TYPE> = <PROP-LIST> */
875 3      /* 69 NO ACTION REQUIRED */

```



```

67          \! EMPTY                                -/
876 3      /* NO ACTION REQUIRED */
68      <PROP-LIST> = <DATA-ELEMENT>                -/
877 3      /* NO ACTION REQUIRED */
69          \! <PROP-LIST> <DATA-ELEMENT>          -/
878 3      /* NO ACTION REQUIRED */
70      <DATA-ELEMENT> = PIC <INPUT>                -/
879 3      CALL PICANALIZER;
71          \! USAGE COMP                            -/
880 3      CALL SET$TYPE<COMP>;
72          \! USAGE DISPLAY                          -/
881 3      /* NO ACTION REQUIRED - DEFAULT */
73          \! SIGN LEADING <SEPARATE>              -/
882 3      CALL SET$SIGN<18>;
74          \! SIGN TRAILING <SEPARATE>              -/
883 3      CALL SET$SIGN<17>;
75          \! OCCURS <INTEGER>                      -/
884 3      DO;
885 4          CALL OR$TYPE<126>;
886 4          CALL SET$OCCURS<VALUE<SP>>;
887 4      END;
76          \! SYNC <DIRECTION>                      -/
888 3      /* NO ACTION REQUIRED - BYTE MACHINE */
77          \! VALUE <LITERAL>                      -/
889 3      DO;
890 4          IF NOT FILE$SEC$END THEN
891 4              DO;
892 5                  CALL PRINT$ERROR<'VE'>;
893 5                  PENDING$LITERAL=0;
894 5              END;
895 4      END;
79      <DIRECTION> = LEFT                            -/
896 3      /* NO ACTION REQUIRED */
79          \! RIGHT                                -/
897 3      /* NO ACTION REQUIRED */
80          \! EMPTY                                -/
898 3      /* NO ACTION REQUIRED */
81      <SEPARATE> = SEPARATE                          -/
899 3      VALUE<SP>=2;
82          \! EMPTY                                -/
900 3      /* NO ACTION REQUIRED */
83      <LITERAL> = <INPUT>                            -/
901 3      DO;
902 4          CALL LOAD$LITERAL;
903 4          PENDING$LITERAL=1;
904 4      END;
84          \! CLITD                                -/
905 3      DO;
906 4          CALL LOAD$LITERAL;
907 4          PENDING$LITERAL=2;
908 4      END;
85          \! ZERO                                -/
909 3      PENDING$LITERAL=3;
86          \! SPACE                                -/
910 3      PENDING$LITERAL=4;
87          \! QUOTE                                -/
911 3      PENDING$LITERAL=5;
88      <INTEGER> = <INPUT>                            -/
912 3      CALL CONVERT$INTEGER;
89      <ID> = <INPUT>                                -/
913 3      VALUE<SP>=MATCH; /* STORE SYMBOL TABLE POINTERS */

914 3      END; /* END OF CASE STATEMENT */
915 2      END CODE$GEN;

916 1      GETIN1 PROCEDURE BYTE;
917 2          RETURN INDEX1<STATE>;
918 2      END GETIN1;

919 1      GETIN2 PROCEDURE BYTE;
920 2          RETURN INDEX2<STATE>;
921 2      END GETIN2;

922 1      INCSP PROCEDURE;
923 2          SP=SP + 1;
924 2          IF SP >= PSTACKSIZE THEN CALL FATAL$ERROR<'SO'>;
925 2          VALUE<SP>=0; /* CLEAR VALUE STACK */
926 2      END INCSP;

928 1      LOOKAHEAD PROCEDURE;
929 2          IF NOLOOK THEN
930 2              DO;
931 3                  CALL SCANNER;
932 3                  NOLOOK=FALSE;
933 3                  IF PRINT$TOKEN THEN
934 3                      DO;

```



```

935 4      CALL CLRF;
936 4      CALL PRINT#NUMBER(TOKEN);
937 4      CALL PRINT#CHAR(' ');
938 4      CALL PRINT#ACCUML;
939 4      END;
940 3      ENO;
941 2      ENO LOOKAHEAD;

942 1      NO$CONFLICT PROCEDURE (CSTATE) BYTE;
943 2      DECLARE (CSTATE, I, J, K) BYTE;
944 2      J=INDEX1(CSTATE);
945 2      K=J + INDEX2(CSTATE) - 1;
946 2      DO I=J TO K;
947 3          IF READ1(I)=TOKEN THEN RETURN TRUE;
948 3      END;
949 2      RETURN FALSE;
950 2      ENO NO$CONFLICT;

952 1      RECOVER PROCEDURE BYTE;
953 2      DECLARE (TSP, RSTATE) BYTE;
954 2      GO FOREVER;
955 3          TSP=SP;
956 3          DO WHILE TSP <> 255;
957 4              IF NO$CONFLICT(RSTATE,#STATESTACK(TSP)) THEN
958 5                  OO, /* STATE WILL READ TOKEN */
959 5                  IF SP<TSP THEN SP = TSP - 1;
960 5                  RETURN RSTATE;
961 5              END;
962 4              TSP = TSP - 1;
963 4          END;
964 3          CALL $SCANNER; /* TRY ANOTHER TOKEN */
965 3      END;
966 2      ENO RECOVER;

968 1      ENO$PASS PROCEDURE;
969 2      /* THIS PROCEDURE STORES THE INFORMATION REQUIRED BY PASS2
970 2      IN LOCATIONS ABOVE THE SYMBOL TABLE. THE FOLLOWING
971 2      INFORMATION IS STORED
972 2      OUTPUT FILE CONTROL BLOCK
973 2      COMPILER TOGGOLES
974 2      INPUT BUFFER POINTER
975 2      THE OUTPUT BUFFER IS ALSO FILLED SO THE CURRENT RECORD IS WRITTEN.
976 2      */
977 2      CALL BYTE$OUT($CO);
978 2      CALL ADDR$OUT(NEXT$AVAILABLE);
979 2      DO WHILE OUTPUT$PTP<> OUTPUT$BUFF;
980 3          CALL BYTE$OUT($OFFH);
981 3      END;

982 2      CALL MOVE(OUTPUT$PCB, MAX$MEMORY-PASS1$LEN, PASS1$LEN);
983 2      L GO TO L; /* PATCH TO "JMP 310BH" */
984 2      ENO$PASS;

/* * * * * PROGRAM EXECUTION STARTS HERE * * */

987 1      CALL MOVE(INITIAL$POS, MAX$MEMORY, ROR$LENGTH);
988 1      CALL INIT$SCANNER;
989 1      CALL INIT$SYMBOL;

/* * * * * * * * * * * PARSE * * * * */

990 1      DO WHILE COMPILING;
991 2      IF STATE <= MAXRNO THEN /* READ STATE */
992 3          DO;
993 4              CALL INCSP;
994 4              STATESTACK(SP) = STATE; /* SAVE CURRENT STATE */
995 4              CALL LOOKAHEAD;
996 4              I=GETIN1;
997 4              J = I + GETIN2 - 1;
998 4              DO I=I TO J;
999 5                  IF READ1(I) = TOKEN THEN
1000 6                      DO;
1001 7                          /* COPY THE ACCUMULATOR IF IT IS AN INPUT
1002 7                          STRING. IF IT IS A RESERVED WORD IT DOES
1003 7                          NOT NEED TO BE COPIED. */
1004 8                          IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
1005 9                              DO K=0 TO ACCUM(0);
1006 10                                  VARC(K)=ACCUM(K);
1007 10                              END;
1008 9                          STATE=READ2(I);
1009 9                          NOLOOK=TRUE;
1010 9                          I=J;
1011 9                      END;
1012 5                  ELSE
1013 5                      IF I=J THEN

```



```

1000 4          DO;
1001 5              CALL PRINT$ERROR('<NP>');
1002 5              CALL PRINT('<< ERROR NEAR $>>');
1003 5              CALL PRINT$ACCUM;
1004 5              IF (STATE =RECOVER)=0 THEN COMPILING=FALSE;
1005 5          END;
1006 5      END;
1007 5  END; /* END OF READ STATE */
1008 5  ELSE
1009 2      IF STATE<MAXPNO THEN /* APPLY PRODUCTION STATE */
1010 2      DO;
1011 3          MP=SP - GETIN2;
1012 3          MP1=MP + 1;
1013 3          CALL CODE$GEN(STATE - MAXPNO);
1014 3          SP=MP;
1015 3          I=GETIN1;
1016 3          J=STATESTACK(SP);
1017 3          DO WHILE (K =APPLY1(I)) <> 0 AND J<OK;
1018 4              I=I + 1;
1019 4          END;
1020 3          IF (K =APPLY2(I))=0 THEN COMPILING=FALSE;
1021 3          STATE=K;
1022 3      END;
1023 5  ELSE
1024 2      IF STATE<MAXLNO THEN /*LOOKAHEAD STATE*/
1025 2      DO;
1026 3          I=GETIN1;
1027 3          CALL LOOKAHEAD;
1028 3          DO WHILE (K =LOOK1(I))<>0 AND TOKEN <OK;
1029 4              I=I+1;
1030 4          END;
1031 3          STATE=LOOK2(I);
1032 5      END;
1033 5  ELSE
1034 2          /*PUSH STATES*/
1035 3          DO;
1036 3              CALL INCSP;
1037 3              STATESTACK(SP)=GETIN2;
1038 3              STATE=GETIN1;
1039 3          END;
1040 2      END; /* OF WHILE COMPILING */
1041 1      CALL CALF;
1042 1      CALL PRINT('<<PROCEDURES>>');
1043 1      CALL END$PASS;
1044 1      END;

```

MODULE INFORMATION

```

CODE AREA SIZE      = 1E91H    7925D
VARIABLE AREA SIZE = 02FCH    764D
MAXIMUM STACK SIZE = 001CH    28D
1517 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

ISIS-II PL/I-88 V3.1 COMPILATION OF MODULE INTERP
 OBJECT MODULE PLACED IN F1 INTERP OBJ
 COMPILER INVOKED BY PLI88 F1 INTERP PLM

```

1      $PAGELENGTH(90)
      INTERP: /* MODULE " I N T E R P " */
          DO,
              /*          COBOL INTERPRETER          */
              /*      NORMALLY ORG'ED TO X'100'      */
              /* GLOBAL DECLARATIONS AND LITERALS */
2      1  DECLARE
          LIT      LITERALLY      'LITERALLY',
          BDOS     LIT      '5H', /* ENTRY TO OPERATING SYSTEM */
          BOOT     LIT      '0',
          CR       LIT      '13',
          LF       LIT      '10',
          TRUE     LIT      '1',
          FALSE    LIT      '0',
          FOREVER   LIT      'WHILE TRUE',
              /* UTILITY VARIABLES */
2      1  DECLARE
          BOOTER    ADDRESS          INITIAL ('0000H'),
          INDEX     BYTE,
          A$CTR     ADDRESS,
          CTR       BYTE,
          BASE      ADDRESS,
          B$BYTE    BASED BASE (1)    BYTE,
          B$ADDR    BASED BASE (1)    ADDRESS,
          HOLD      ADDRESS,
          H$BYTE    BASED HOLD (1)    BYTE,
          H$ADDR    BASED HOLD (1)    ADDRESS,
              /* CODE POINTERS */
          CODE$START LIT      '2000H',
          PROGRAM$COUNTER ADDRESS,
          C$BYTE     BASED PROGRAM$COUNTER (1)  BYTE,
          C$ADDR     BASED PROGRAM$COUNTER (1)  ADDRESS,
              /* * * * * GLOBAL INPUT AND OUTPUT ROUTINES * * * * */
4      1  DECLARE
          CURRENT$FCS ADDRESS,
          START$OFFSET LIT      '36',
5      1  MON1. PROCEDURE (F,A) EXTERNAL,
6      2  DECLARE F BYTE, A ADDRESS,
7      2  END MON1,
9      1  MON2. PROCEDURE (F,A) BYTE EXTERNAL,
9      2  DECLARE F BYTE, A ADDRESS,
10     2  END MON2,
11     1  PRINT$CHAR. PROCEDURE (CHAR),
12     2  DECLARE CHAR BYTE,
13     2  CALL MON1 (2,CHAR),
14     2  END PRINT$CHAR,
15     1  CRLF. PROCEDURE,
16     2  CALL PRINT$CHAR(CR),
17     2  CALL PRINT$CHAR(LF),
18     2  END CRLF,
19     1  PRINT. PROCEDURE (A),
20     2  DECLARE A ADDRESS,
21     2  CALL CRLF,
22     2  CALL MON1(9,A),
23     2  END PRINT,
24     1  READ. PROCEDURE(A),
25     2  DECLARE A ADDRESS,
26     2  CALL MON1(10,A),
27     2  END READ,

```



```

28 1 PRINT$ERROR PROCEDURE (CODE),
29 2 DECLARE CODE ADDRESS,
30 2 CALL CRLF,
31 2 CALL PRINT$CHAR(HIGH(CODE)),
32 2 CALL PRINT$CHAR(LOW(CODE)),
33 2 END PRINT$ERROR;

34 1 FATAL$ERROR PROCEDURE(CODE),
35 2 DECLARE CODE ADDRESS,
36 2 CALL PRINT$ERROR(CODE),
37 2 CALL $DOSTER,
38 2 END FATAL$ERROR;

39 1 SET$DMA: PROCEDURE,
40 2 CALL MON1 (26, CURRENT$FCB + START$OFFSET),
41 2 END SET$DMA;

42 1 OPEN PROCEDURE (ADDR) BYTE,
43 2 DECLARE ADDR ADDRESS,
44 2 CALL SET$DMA; /* INSURE DIRECTORY READ WON'T CLOBBER CODE */
45 2 RETURN MON2(15, ADDR),
46 2 END OPEN;

47 1 CLOSE PROCEDURE (ADDR),
48 2 DECLARE ADDR ADDRESS,
49 2 IF MON2(16, ADDR)=255 THEN CALL FATAL$ERROR('CL'),
51 2 END CLOSE;

52 1 DELETE: PROCEDURE,
53 2 CALL MON1(19, CURRENT$FCB),
54 2 END DELETE;

55 1 MAKE: PROCEDURE (ADDR),
56 2 DECLARE ADDR ADDRESS,
57 2 IF MON2(22, ADDR)=255 THEN CALL FATAL$ERROR('ME'),
59 2 END MAKE;

60 1 DISK$READ: PROCEDURE BYTE,
61 2 RETURN MON2(20, CURRENT$FCB),
62 2 END DISK$READ;

63 1 DISK$WRITE: PROCEDURE BYTE,
64 2 RETURN MON2(21, CURRENT$FCB),
65 2 END DISK$WRITE;

/* * * * * * UTILITY PROCEDURES * * * * * */

66 1 DECLARE
SUBSCRIPT (S) ADDRESS,

67 1 RES PROCEDURE(ADDR) ADDRESS,
/* THIS PROCEDURE RESOLVES THE ADDRESS OF A SUBSCRIPTED
IDENTIFIER OR A LITERAL CONSTANT */

68 2 DECLARE ADDR ADDRESS,
69 2 IF ADDR > 32 THEN RETURN ADDR,
71 2 IF ADDR < 9 THEN RETURN SUBSCRIPT(ADDR),
73 2 DO CASE ADDR = 9:
74 3 RETURN ('0'),
75 3 RETURN (' '),
76 3 RETURN (' '),
77 3 END,
79 2 RETURN 0;
79 2 END RES;

80 1 MOVE PROCEDURE(FROM, DESTINATION, COUNT),
81 2 DECLARE (FROM, DESTINATION, COUNT) ADDRESS,
/* BASED FROM, 0 BASED DESTINATION, BYTE,
82 2 DO WHILE (COUNT = COUNT - 1) < 0FFFFH,
83 3 0=F,
84 3 FROM=FROM + 1,
85 3 DESTINATION=DESTINATION + 1,
86 1 END;

```



```

87 2      END MOVE;

88 1      FILL: PROCEDURE(DESTINATION, COUNT, CHAR);
89 2          DECLARE (DESTINATION, COUNT) ADDRESS;

              (CHAR, 0 BASED DESTINATION) BYTE;
          DO WHILE (COUNT = COUNT - 1) <> 0FFFFH;
90 2              DO CHAR;
91 3                  D=CHAR;
92 3                  DESTINATION=DESTINATION + 1;
93 3              END;
94 2      END FILL;

95 1      CONVERT$TO$HEX: PROCEDURE(POINTER, COUNT) ADDRESS;
96 2          DECLARE POINTER ADDRESS; COUNT BYTE;
97 2          A$CTR=0;
98 2          BASE=POINTER;
99 2          DO CTR = 0 TO COUNT-1;
100 3              A$CTR=SHL(A$CTR, 2) + SHL(A$CTR, 1) + B$BYTE(CTR) - '0';
101 3          END;
102 2          RETURN A$CTR;
103 2      END CONVERT$TO$HEX;

/* * * * * * * * * * * CODE CONTROL PROCEDURES * * * * * */

104 1      DECLARE
          BRANCH$FLAG          BYTE          INITIAL(FALSE);

105 1      INC$PTR: PROCEDURE (COUNT);
106 2          DECLARE COUNT BYTE;
107 2          PROGRAM$COUNTER=PROGRAM$COUNTER + COUNT;
108 2      END INC$PTR;

109 1      GET$OP$CODE: PROCEDURE BYTE;
110 2          CTR=C$BYTE(0);
111 2          CALL INC$PTR(1);
112 2          RETURN CTR;
113 2      END GET$OP$CODE;

114 1      COND$BRANCH: PROCEDURE(COUNT);
115 2          /* THIS PROCEDURE CONTROLS BRANCHING INSTRUCTIONS */
116 2          DECLARE COUNT BYTE;
117 2          IF BRANCH$FLAG THEN
118 3              DO;
119 3                  BRANCH$FLAG=FALSE;
120 3                  PROGRAM$COUNTER=C$ADDR(COUNT);
121 3              END;
122 2          ELSE CALL INC$PTR(SHL(COUNT, 1)*2);
123 2      END COND$BRANCH;

123 1      INCR$OP$BRANCH: PROCEDURE(MARK);
124 2          DECLARE MARK BYTE;
125 2          IF MARK THEN CALL INC$PTR(2);
126 2          ELSE PROGRAM$COUNTER=C$ADDR(0);
127 2      END INCR$OP$BRANCH;

/* * * * * * * * * * * COMPARISONS * * * * * */

129 1      CHAR$COMPARE: PROCEDURE BYTE;
130 2          BASE=C$ADDR(0);
131 2          HOLD=C$ADDR(1);
132 2          DO A$CTR=0 TO C$ADDR(2) - 1;
133 3              IF B$BYTE(A$CTR) > H$BYTE(A$CTR) THEN RETURN 1;
134 3              IF B$BYTE(A$CTR) < H$BYTE(A$CTR) THEN RETURN 0;
135 3          END;
136 2          RETURN 2;
137 2      END CHAR$COMPARE;

140 1      STRING$COMPARE: PROCEDURE(PIVOT);
141 2          DECLARE PIVOT BYTE;
142 2          IF CHAR$COMPARE=PIVOT THEN BRANCH$FLAG=NOT BRANCH$FLAG;
143 2          CALL COND$BRANCH(0);
144 2      END STRING$COMPARE;

146 1      NUMERIC: PROCEDURE(CHAR) BYTE;
147 2          DECLARE CHAR BYTE;

```



```

148 2      RETURN (CHAR >='0') AND (CHAR <='9'),
149 2      END NUMERIC;

150 1      LETTER PROCEDURE(CHAR) BYTE,
151 2      DECLARE CHAR BYTE,
152 2      RETURN (CHAR >='A') AND (CHAR <='Z'),
153 2      END LETTER;

154 1      SIGN PROCEDURE(CHAR) BYTE,
155 2      DECLARE CHAR BYTE,
156 2      RETURN (CHAR='+') OR (CHAR='-'),
157 2      END SIGN;

158 1      COMP$NUM$UNSIGNED: PROCEDURE,
159 2      BASE=C$ADDR(0),
160 2      DO A$CTR=0 TO C$ADDR(2)-1,
161 3      IF NOT NUMERIC(B$BYTE(A$CTR)) THEN
162 3      DO,
163 4      BRANCH$FLAG=NOT BRANCH$FLAG,
164 4      RETURN,
165 4      END,
166 3      END,
167 2      CALL COND$BRANCH(2),
168 2      END COMP$NUM$UNSIGNED;

169 1      COMP$NUM$SIGN: PROCEDURE,
170 2      BASE=C$ADDR(0),
171 2      DO A$CTR=0 TO C$ADDR(2)-1,
172 3      IF NOT (NUMERIC(CTR=B$BYTE(A$CTR))
173 3      OR SIGN(CTR)) THEN
174 4      BRANCH$FLAG=NOT BRANCH$FLAG,
175 4      RETURN,
176 4      END,
177 3      END,
178 2      CALL COND$BRANCH(2),
179 2      END COMP$NUM$SIGN;

180 1      COMP$ALPHA PROCEDURE,
181 2      BASE=C$ADDR(0),
182 2      DO A$CTR=0 TO C$ADDR(2)-1,
183 3      IF NOT LETTER(B$BYTE(A$CTR)) THEN
184 3      DO,
185 4      BRANCH$FLAG=NOT BRANCH$FLAG,
186 4      RETURN,
187 4      END,
188 3      END,
189 2      CALL COND$BRANCH(2),
190 2      END COMP$ALPHA;

/* * * * * * * * * * * NUMERIC OPERATIONS * * * * * * * * * */

191 1      DECLARE

      (R0,R1,R2)      (10)      BYTE, /* REGISTERS */
      SIGN(1)          BYTE,
      (DEC$PT0,DEC$PT1,DEC$PT2)  BYTE,
      DEC$PTA (3)      BYTE AT ( DEC$PT0),
      OVERFLOW          BYTE,
      R$PTR             BYTE,
      SWITCH            BYTE,
      SIGNIF$NO         BYTE,
      ZONE              LIT      '10H',
      POSITIVE          LIT      '1',
      NEGITIVE          LIT      '0',

192 1      CHECK$FOR$SIGN PROCEDURE(CHAR) BYTE,
193 2      DECLARE CHAR BYTE,
194 2      IF NUMERIC(CHAR) THEN RETURN POSITIVE,
195 2      IF NUMERIC(CHAR - ZONE) THEN RETURN NEGITIVE,
196 2      CALL PRINT$ERROR('SI'),
197 2      RETURN POSITIVE,
198 2      END CHECK$FOR$SIGN;

199 1      STORE$IMMEDIATE PROCEDURE,
200 2      DO CTR=0 TO 3,
201 3      R0(CTR)=R2(CTR),

```



```

204 1      END.
205 2      DEC$PT0=DEC$PT2;
206 3      SIGN0(0)=SIGN0(2);
207 2      END STORE$IMMEDIATE.

208 1      ONE$LEFT PROCEDURE;
209 2      DECLARE (CTR, FLAG) BYTE;
210 2      IF ~(FLAG=SHR(B$BYTE(0),4))=0) OR (FLAG=9) THEN
211 2      DO;
212 3          DO CTR=0 TO 8;
213 4              B$BYTE(CTR)=SHL(B$BYTE(CTR),4) OR SHR(B$BYTE(CTR + 1),4);
214 4          END;
215 3          B$BYTE(9)=SHL(B$BYTE(9),4) OR FLAG;
216 3      END;
217 2      ELSE OVERFLOW=TRUE;
218 2      END ONE$LEFT.

219 1      ONE$RIGHT PROCEDURE;
220 2      DECLARE CTR BYTE;
221 2      CTR=10;
222 2      DO INDEX=1 TO 9;
223 3          CTR=CTR-1;
224 3          B$BYTE(CTR)=SHR(B$BYTE(CTR),4) OR SHL(B$BYTE(CTR-1),4);
225 3      END;
226 2      B$BYTE(0)=SHR(B$BYTE(0),4);
227 2      IF B$BYTE(0) = 09H THEN
228 2          B$BYTE(0) = 99H;
229 2      END ONE$RIGHT;

230 1      SHIFT$RIGHT PROCEDURE(COUNT);
231 2      DECLARE COUNT BYTE;
232 2      DO CTR=1 TO COUNT;
233 3          CALL ONE$RIGHT;
234 3      END;
235 2      END SHIFT$RIGHT;

236 1      SHIFT$LEFT PROCEDURE(COUNT);
237 2      DECLARE COUNT BYTE;
238 2      OVERFLOW=FALSE;
239 2      DO CTR=1 TO COUNT;
240 3          CALL ONE$LEFT;
241 3          IF OVERFLOW THEN RETURN;
242 3      END;
243 2      END SHIFT$LEFT;

244 2      END SHIFT$LEFT;

245 1      ALIGN PROCEDURE;
246 2      BASE= 00;
247 2      IF DEC$PT0 > DEC$PT1 THEN CALL SHIFT$RIGHT(DEC$PT0-DEC$PT1);
248 2      ELSE CALL SHIFT$LEFT(DEC$PT1-DEC$PT0);
249 2      END ALIGN.

250 1      ADD$0 PROCEDURE(SECOND, DEST);
251 2      DECLARE (SECOND, DEST) ADDRESS, (CY, A, B, I, J) BYTE;
252 2      HOLD= SECOND;
253 2      BASE = DEST;
254 2      CY=0;
255 2      CTR=9;
256 2      DO J=1 TO 10;
257 3          A=0(CTR);
258 3          B=A$BYTE(CTR);
259 3          I=DEC(A+CY);
260 3          CY=CARRY;
261 3          I=DEC(I + B);
262 3          CY=(CY OR CARRY) AND 1;
263 3          B$BYTE(CTR)=I;
264 3          CTR=CTR-1;
265 3      END;
266 2      IF CY THEN
267 3          DO;
268 4              CTR=9;
269 4              DO J = 1 TO 10;
270 5                  I=B$BYTE(CTR);
271 5                  I=DEC(I+CY);
272 5                  CY=CARRY AND 1;
273 5                  B$BYTE(CTR)=I;
274 5                  CTR=CTR-1;
275 5              END;
276 4          END;
277 3      END;
278 2      END ADD$0;

```



```

273 1  COMPLIMENT PROCEDURE(NUMB).
280 2  DECLARE NUMB BYTE.

281 2      SIGN0(NUMB) = SIGN0(NUMB) XOR 1.  /* COMPLIMENT SIGN */

282 2  DO CASE NUMB.
283 3      HOLD= R0.
284 3      HOLD= R1.
285 3      HOLD= R2.
286 3  END.

287 2  DO CTR=0 TO 9.
288 3      H$BYTE(CTR)=29H - H$BYTE(CTR).
289 3  END.

290 2  END COMPLIMENT.

291 1  R2$ZERO PROCEDURE BYTE.
292 2  DECLARE I BYTE.
293 2  IF (SHL(R2(0),4)<0) OR (SHR(R2(9),4)<0)
294 3  THEN RETURN FALSE.
295 2  ELSE DO I=1 TO 8.
296 3      IF R2(I)<0 THEN RETURN FALSE.
297 3  END.
298 2  RETURN TRUE.
299 2  END R2$ZERO.

301 1  CHECK$RESULT PROCEDURE.
302 2  IF SHR(R2(0),4)=9 THEN CALL COMPLIMENT(2).
303 2  IF SHR(R2(0),4)<0 THEN OVERFLOW=TRUE.
304 2  END CHECK$RESULT.

307 1  CHECK$SIGN PROCEDURE.
308 2  IF SIGN0(0) AND SIGN0(1) THEN
309 3  DO.
310 4      SIGN0(2)=POSITIVE.
311 4      RETURN.
312 3  END.
313 2  SIGN0(2)=NEGATIVE.
314 2  IF NOT SIGN0(0) AND NOT SIGN0(1) THEN RETURN.
315 2  IF SIGN0(0) THEN CALL COMPLIMENT(1).
316 2  ELSE CALL COMPLIMENT(0).
317 2  END CHECK$SIGN.

320 1  LEADING$ZEROS PROCEDURE (ADDR) BYTE.
321 2  DECLARE COUNT BYTE, ADDR ADDRESS.
322 2  COUNT=0.
323 2  BASE=ADDR.
324 2  DO CTR=0 TO 9.
325 3  IF (B$BYTE(CTR) AND 0F0H) < 0 THEN RETURN COUNT.
326 3  COUNT=COUNT + 1.
327 3  IF (B$BYTE(CTR) AND 0FH) < 0 THEN RETURN COUNT.
328 3  COUNT=COUNT + 1.
329 3  END.
330 2  RETURN COUNT.
331 2  END LEADING$ZEROS.

334 1  CHECK$DECIMAL PROCEDURE.
335 2  IF DEC$T2<(CTR=C$BYTE(2)) THEN
336 3  DO.
337 4      BASE= R2.
338 4      IF DEC$T2 > CTR THEN CALL SHIFT$RIGHT(DEC$T2-CTR).
339 4      ELSE CALL SHIFT$LEFT(CTR-DEC$T2).
340 3  END.
341 2  IF LEADING$ZEROS( R2 ) < 19 - C$BYTE(2) THEN OVERFLOW = TRUE.
342 2  END CHECK$DECIMAL.

345 1  ADD PROCEDURE.
346 2  OVERFLOW=FALSE.
347 2  CALL ALIGN.
348 2  CALL CHECK$SIGN.
349 2  CALL ADDR0( P1, R2).
350 2  CALL CHECK$RESULT.
351 2  END ADD.

352 1  ADD$SERIES PROCEDURE(COUNT).
353 2  DECLARE (I,COUNT) BYTE.
354 2  DO I=1 TO COUNT.
355 3  CALL ADD$R0( R2, R2).

```



```

136 1      END,
137 2      END ADD$SERIES.

158 1      SET$MULT$DIV. PROCEDURE,
159 2      OVERFLOW=FALSE,
160 2      SIGN$K2 = (NOT (SIGN$K0) XOR SIGN$K1)) AND 01H,
161 2      CALL FILL$R2,10,0),
162 2      END SET$MULT$DIV.

163 1      R1$GREATER. PROCEDURE BYTE,
164 2      DECLARE I BYTE,
165 2      DO CTR=0 TO 9,
166 2      IF R1(CTR)>I = 99H-R0(CTR) THEN RETURN TRUE,
167 2      IF R1(CTR)<I THEN RETURN FALSE,
168 2      END,
169 2      RETURN TRUE,
170 2      END R1$GREATER.

173 1      MULTIPLY. PROCEDURE(VALUE),
174 2      DECLARE VALUE BYTE,
175 2      IF VALUE<0 THEN CALL ADD$SERIES(VALUE),
176 2      BASE=R0,
177 2      CALL ONE$LEFT,
178 2      END MULTIPLY.

180 1      DIVIDE. PROCEDURE,
181 2      DECLARE (I, J, K, LZ0, LZ1, X) BYTE,
182 2      CALL SET$MULT$DIV,
183 2      IF (LZ0 = LEADING$ZEROS$R0) AND
184 2      (X = (LZ1 = LEADING$ZEROS$R1)) THEN
185 2      DO,
186 2      IF LZ0>LZ1 THEN
187 4      DO,
188 4      BASE = R0,
189 4      CALL SHIFT$LEFT(I = LZ0-LZ1),
190 4      DECP$T0=DECP$T0 + I,
191 4      X = LZ1,
192 4      END,
193 4      ELSE DO,
194 4      BASE = R1,
195 4      CALL SHIFT$LEFT(I = LZ1-LZ0),
196 4      DECP$T1=DECP$T1 + I,
197 4      X = LZ0,
198 4      END,
199 2      DECP$T2= 16 - X + DECP$T1 - DECP$T0,
200 2      CALL COMPLIMENT(0),
201 2      DO I = X TO 15,
202 2      J=0,
203 2      DO WHILE R1$GREATER,
204 4      CALL ADD$R0(R1, R1),
205 4      IF R1(0) = 99H THEN
206 4      CALL COMPLIMENT(1),
207 4      J=J+1,
208 4      END,
209 2      K=SHR(I,1),
210 2      IF I THEN R2(K)=R2(K) OR J,
211 2      ELSE R2(K)=R2(K) OR SHL(J,4),
212 2      BASE=R0,
213 2      CALL ONE$RIGHT,
214 2      END,
215 2      END DIVIDE.

417 1      LOAD$A$CHAR. PROCEDURE(CHAR),
418 2      DECLARE CHAR BYTE,
419 2      IF (SWITCH =NOT SWITCH) THEN
420 2      B$BYTE($PTR)=B$BYTE($PTR) OR SHL(CHAR - 10H,4),
421 2      ELSE B$BYTE($PTR = $PTR-1)=CHAR - 10H,
422 2      END LOAD$A$CHAR.

423 1      LOAD$NUMBERS. PROCEDURE(ADDR, CNT),
424 2      DECLARE ADDR ADDRESS, (I, CNT) BYTE,
425 2      HOLD=RES(ADDR),
426 2      CTR=CNT,
427 2      DO INDEX = 1 TO CNT,
428 2      CTR=CTR-1,
429 2      CALL LOAD$A$CHAR(H$BYTE(CTR)),
430 2      END,
431 2      CALL INC$PTR(3),

```



```

432 2      END LOAD$NUMBERS;

433 1      SET$LOAD PROCEDURE (SIGN$IN);
434 2      DECLARE SIGN$IN BYTE;
435 2      DO CASE (CTR = C$BYTE(4));
436 3          BASE = R0;
437 3          BASE = R1;
438 3          BASE = R2;
439 3      END;
440 2      DEC$PTA(CTR) = C$BYTE(3);
441 2      SIGN$(CTR) = SIGN$IN;
442 2      CALL FILL (BASE, 10, 0);
443 2      R$PTR = 9;
444 2      SWITCH = FALSE;
445 2      END SET$LOAD;

446 1      LOAD$NUMERIC PROCEDURE;
447 2      CALL SET$LOAD(1);
448 2      CALL LOAD$NUMBERS(C$ADDR(0), C$BYTE(2));
449 2      END LOAD$NUMERIC;

450 1      LOAD$NUM$LIT PROCEDURE;
451 2      DECLARE (LIT$SIZE, FLAG) BYTE;

452 2      CHAR$SIGN PROCEDURE;
453 3      LIT$SIZE = LIT$SIZE - 1;
454 3      HOLD = HOLD + 1;
455 3      END CHAR$SIGN;

456 2      LIT$SIZE = C$BYTE(2);
457 2      HOLD = C$ADDR(0);
458 2      IF H$BYTE(0) = '-' THEN
459 3      DO;
460 4          CALL CHAR$SIGN;
461 4          CALL SET$LOAD(NEGATIVE);
462 4      END;
463 2      ELSE DO;
464 3      IF H$BYTE(0) = '+' THEN CALL CHAR$SIGN;
465 3      CALL SET$LOAD(POSITIVE);
466 3      END;
467 2      FLAG = 0;
468 2      CTR = LIT$SIZE;
469 2      DO INDEX = 1 TO LIT$SIZE;
470 3      CTR = CTR - 1;
471 3      IF H$BYTE(CTR) = '-' THEN FLAG = LIT$SIZE - (CTR + 1);
472 3      ELSE CALL LOAD$AS$CHAR(H$BYTE(CTR));
473 3      END;
474 2      DEC$PTA(C$BYTE(4)) = FLAG;
475 2      CALL INC$PTR(3);
476 2      END LOAD$NUM$LIT;

477 1      STORE$ONE PROCEDURE;
478 2      IF (SWITCH = NOT SWITCH) THEN
479 3      B$BYTE(0) = SHR(H$BYTE(0), 4) OR '0';
480 3      ELSE DO;
481 4      HOLD = HOLD - 1;
482 4      B$BYTE(0) = (H$BYTE(0) AND 0FH) OR '0';
483 4      END;
484 2      BASE = BASE - 1;
485 2      END STORE$ONE;

486 1      STORE$AS$CHAR PROCEDURE (COUNT);
487 2      DECLARE COUNT BYTE;
488 2      SWITCH = FALSE;
489 2      HOLD = R2 + 3;
490 2      DO CTR = 1 TO COUNT;
491 3      CALL STORE$ONE;
492 3      END;
493 2      END STORE$AS$CHAR;

494 1      SET$ZONE PROCEDURE (ADDR);
495 2      DECLARE ADDR ADDRESS;
496 2      IF NOT SIGN$(2) THEN
497 3      DO;
498 4      BASE = ADDR;
499 4      B$BYTE(0) = B$BYTE(0) OR ZONE;
500 4      END;
501 2      CALL INC$PTR(4);
502 2      END SET$ZONE;

```



```

505 1  SET$SIGN$SEP  PROCEDURE (ADDR);
506 2  DECLARE ADDR ADDRESS;
507 2  BASE=ADDR;
508 2  IF SIGN(2) THEN B$BYTE(0)='+';
509 2  ELSE B$BYTE(0)='-';
510 2  CALL INC$PTR(4);
511 2  END SET$SIGN$SEP;

512 2

513 1  STORE$NUMERIC PROCEDURE;
514 2  CALL CHECK$DECIMAL;
515 2  BASE=C$ADDR(0) + C$BYTE(2) -1;
516 2  CALL STORE$AS$CHAR(C$BYTE(2));
517 2  END STORE$NUMERIC;

/* + + + + + INPUT-OUTPUT ACTIONS + + + + + */

518 1  DECLARE
      FLAG$OFFSET      LIT      '33';
      EXTENT$OFFSET    LIT      '12';
      REC$NO           LIT      '32';
      PTR$OFFSET       LIT      '17';
      BUFF$LENGTH      LIT      '128';
      VAR$END          LIT      'CR';
      TERMINATOR       LIT      'LAH';
      END$OF$RECORD    BYTE;
      INVALID          BYTE;
      RANDOM$FILE      BYTE;
      CURRENT$FLAG     BYTE;
      FCB$BYTE         BASED CURRENT$FCB      BYTE;
      FCB$ADDR         BASED CURRENT$FCB      ADDRESS;
      FCB$BYTE$A       BASED CURRENT$FCB (1) BYTE;
      FCB$ADDR$A       BASED CURRENT$FCB (1) ADDRESS;
      BUFF$PTR         ADDRESS;
      BUFF$END         ADDRESS;
      BUFF$START       ADDRESS;
      BUFF$BYTE        BASED BUFF$PTR      BYTE;
      CON$BUFF         ADDRESS INITIAL (20H);
      CON$BYTE         BASED CON$BUFF      BYTE;
      CON$INPUT        ADDRESS INITIAL (32H);

519 1  ACCEPT  PROCEDURE;
520 2  CALL CALF;
521 2  CALL PRINT$CHAR(3FH);
522 2  /* CALL CALF; */
523 2  CALL FILL(CON$INPUT, (CON$BYTE=C$BYTE(2), ' ');
524 2  CALL READ(CON$BUFF);
525 2  CALL MOVE(CON$INPUT, RES(C$ADDR(0)), CON$BYTE);
526 2  CALL INC$PTR(2);
527 2  END ACCEPT;

527 1  DISPLAY PROCEDURE;
528 2  DECLARE $CNT BYTE BLANK LIT 20H;
529 2  BASE=C$ADDR(0);
530 2  CALL CALF;
531 2  $CNT = C$BYTE(2);
532 2  DO WHILE
533 3  B$BYTE($CNT = $CNT - 1) = BLANK;
534 2  DO CTR = 0 TO $CNT;
535 3  CALL PRINT$CHAR(B$BYTE(CTR));
536 3  END;
537 2  CALL INC$PTR(2);
538 2  END DISPLAY;

539 1  SET$FILE$TYPE PROCEDURE (TYPE);
540 2  DECLARE TYPE BYTE;
541 2  BASE=C$ADDR(0);
542 2  B$BYTE(FLAG$OFFSET)=TYPE;
543 2  END SET$FILE$TYPE;

544 1  GET$FILE$TYPE PROCEDURE BYTE;
545 2  BASE=C$ADDR(0);
546 2  RETURN B$BYTE(FLAG$OFFSET);
547 2  END GET$FILE$TYPE;

```



```

548 1  SET$I$O  PROCEDURE,
549 2  END$OF$RECORD, INVALID$=FALSE,
550 3  IF C$ADDR(0)=CURRENT$PCB THEN RETURN,
    /* STORE CURRENT POINTERS AND SET INTERNAL WRITE MARK */
551 2  BASE$=CURRENT$PCB,
552 2  PCB$ADDR$A(CTR$OFFSET)=BUFF$PTR,
553 2  PCB$BYTE$A(FLAG$OFFSET)=CURRENT$FLAG,
554 2  /* LOAD NEW VALUES */
555 2  BUFF$END=(BUFF$START+(CURRENT$PCB=C$ADDR(0))/=START$OFFSET)
    + BUFF$LENGTH,
556 2  CURRENT$FLAG=PCB$BYTE$A(FLAG$OFFSET),
557 2  BUFF$PTR=PCB$ADDR$A(CTR$OFFSET),
558 2  END SET$I$O.

559 1  OPEN$FILE  PROCEDURE(TYPE),
560 2  DECLARE TYPE BYTE,
561 2  CALL SET$FILE$TYPE(TYPE),
562 2  CTR=OPEN(CURRENT$PCB=C$ADDR(0)),
563 2  DO CASE TYPE=1
    /* INPUT */
564 1  DO,
565 4  IF CTR=255 THEN CALL PRINT$ERROR( 'NF' ),
566 4  PCB$ADDR$A(CTR$OFFSET)=CURRENT$PCB+100H,
567 4  END,
    /* OUTPUT */
568 4  DO,
569 4  CALL DELETE,
570 4  CALL MAKE(C$ADDR(0)),
571 4  PCB$ADDR$A(CTR$OFFSET)=CURRENT$PCB+START$OFFSET-1,
572 4  END,
    /* I-O */
573 4  DO,
574 4  IF CTR=255 THEN CALL FATAL$ERROR( 'NF' ),
575 4  PCB$ADDR$A(CTR$OFFSET)=CURRENT$PCB + 100H,
576 4  END,
577 1  END,
578 2  CURRENT$PCB=0, /* FORCE A PARAMETER LOAD */
579 2  CALL SET$I$O,
580 2  CALL INC$PTR(2),
581 2  END OPEN$FILE.

584 1  WRITE$MARK, PROCEDURE BYTE,
585 2  RETURN POL(CURRENT$FLAG,1),
586 2  END WRITE$MARK.

587 1  SET$WRITE$MARK  PROCEDURE,
588 2  CURRENT$FLAG=CURRENT$FLAG OR 80H,
589 2  END SET$WRITE$MARK.

590 1  WRITE$RECORD, PROCEDURE,
591 2  IF NOT $WR(CURRENT$FLAG,1) THEN CALL FATAL$ERROR( 'WI' ),
592 2  CALL SET$DMA,
593 2  CURRENT$FLAG=CURRENT$FLAG AND 0FH,
594 2  IF (CTR #DISK$READ) #0 THEN RETURN,
595 2  INVALID$=TRUE,
596 2  END WRITE$RECORD.

599 1  READ$RECORD  PROCEDURE,
600 2  CALL SET$DMA,
601 2  IF WRITE$MARK THEN CALL WRITE$RECORD,
602 2  IF (CTR #DISK$READ) #0 THEN RETURN,
603 2  IF CTR=1 THEN END$OF$RECORD$=TRUE,
604 2  ELSE INVALID$=TRUE,
605 2  END READ$RECORD.

609 1  READ$BYTE  PROCEDURE BYTE,
610 2  IF (BUFF$PTR #BUFF$PTR + 1) >= BUFF$END THEN
611 2  DO,
612 2  CALL READ$RECORD,
613 2  IF END$OF$RECORD THEN RETURN TERMINATOR,
614 2  BUFF$PTR=BUFF$START,
615 2  END,
616 2  RETURN BUFF$BYTE,
617 2  END READ$BYTE.

619 1  WRITE$BYTE  PROCEDURE (CHAR),
620 2  DECLARE CHAR BYTE,
621 2  IF (BUFF$PTR #BUFF$PTR + 1) >= BUFF$END THEN
622 2  DO,

```



```

624 1          BUFFPTR=BUFFSTART,
625 1      END,
626 2      CALL SET$WRITE$MARK,
627 2      BUFF$BYTE=CHAR,
628 2      END WRITE$BYTE,

629 1      WRITE$END$MARK PROCEDURE,
630 2      CALL WRITE$BYTE(CTR),
631 2      CALL WRITE$BYTE(EOF),
632 2      END WRITE$END$MARK,

633 1      READ$END$MARK PROCEDURE,
634 2      IF READ$BYTE(CTR) THEN CALL PRINT$ERROR('EM '),
635 2      IF READ$BYTE(EOF) THEN CALL PRINT$ERROR('EM '),
636 2      END READ$END$MARK,

637 1      READ$VARIABLE PROCEDURE,
638 2      CALL SET$IO,
639 2      BASE=C$ADDR(1),
640 2      DO A$CTR=0 TO C$ADDR(2)-1,
641 3          IF (CTR=(C$BYTE(A$CTR)-READ$BYTE))=VAR$END THEN
642 4              DO,
643 5                  CTR=READ$BYTE,
644 5                  RETURN,
645 4              END,
646 3          IF CTR=TERMINATOR THEN
647 4              DO,
648 5                  END$OF$RECORD=TRUE,
649 5                  RETURN,
650 4              END,
651 3          END,
652 2      CALL READ$END$MARK,
653 2      END READ$VARIABLE,

654 1      WRITE$VARIABLE PROCEDURE,
655 2      DECLARE COUNT ADDRESS,
656 2      CALL SET$IO,
657 2      BASE=C$ADDR(1),
658 2      COUNT=C$ADDR(2),
659 2      DO WHILE(C$BYTE(COUNT)=COUNT-1) AND (COUNT<0),
660 3          END,
661 3          DO A$CTR=0 TO COUNT,
662 4              CALL WRITE$BYTE(C$BYTE(A$CTR)),
663 4              END,
664 3          CALL WRITE$END$MARK,
665 2      END WRITE$VARIABLE,

666 1      READ$TO$MEMORY PROCEDURE,
667 2      CALL SET$IO,
668 2      BASE=C$ADDR(1),
669 2      DO A$CTR=0 TO C$ADDR(2)-1,
670 3          IF (C$BYTE(A$CTR)-READ$BYTE)=TERMINATOR THEN
671 4              DO,
672 5                  END$OF$RECORD=TRUE,
673 5                  RETURN,
674 4              END,
675 3          END,
676 2      CALL READ$END$MARK,
677 2      END READ$TO$MEMORY,

678 1      WRITE$FROM$MEMORY PROCEDURE,
679 2      CALL SET$IO,
680 2      BASE=C$ADDR(1),
681 2      DO A$CTR=0 TO C$ADDR(2)-1,
682 3          CALL WRITE$BYTE(C$BYTE(A$CTR)),
683 3          END,
684 2      CALL WRITE$END$MARK,
685 2      END WRITE$FROM$MEMORY,

/* * * * * * RANDOM I-O PROCEDURES * * * * * */

686 1      SET$RANDOM$POINTER PROCEDURE,
687 2      /*
688 3      THIS PROCEDURE READS THE RANDOM KEY AND COMPUTES
689 3      WHICH RECORD NEEDS TO BE AVAILABLE IN THE BUFFER
690 3      THAT RECORD IS MADE AVAILABLE AND THE POINTERS
691 3      SET FOR INPUT OR OUTPUT
692 3      */
693 2      DECLARE (C$BYTE)COUNT RECORD$ ADDRESS,

```



```

690 2      ENTENT BYTE;
691 2      CALL SET#I#0;
692 2      BYTE#COUNT=(C#ADDR(2)+1)*CONVERT#TO#HEX(C#ADDR(2),C#BYTE(6));
693 2      RECORD#SHR#BYTE#COUNT,7;
694 2      ENTENT#SHR#RECORD,7;
695 2      IF ENTENT<FCB#BYTE#A#ENTENT#OFFSET THEN
696 3      DO;
697 3      IF WRITE#MARK THEN CALL WRITE#RECORD;
698 3      CALL CLOSE(C#ADDR(0));
699 3      FCB#BYTE#A#ENTENT#OFFSET#ENTENT;
700 3      IF OPEN(C#ADDR(0))<0 THEN
701 4      DO;
702 4      IF SHR#CURRENT#FLAG,1 THEN CALL MAKE(C#ADDR(0));
703 4      ELSE INVALID#TRUE;
704 4      END;
705 3      END;
706 2      ENCL;
707 2      BUFF#PTR#(BYTE#COUNT AND 7FH) + BUFF#START -1;
708 2      IF FCB#BYTE#A#REC#NO<0<<CTR #LOW#RECORD#AND 7FH THEN
709 3      DO;
710 3      FCB#BYTE#A(2)=CTR;
711 3      CALL READ#RECORD;
712 3      END;
713 2      END SET#RANDOM#POINTER;

714 1      GET#REC#NUMBER PROCEDURE;
715 2      DECLARE (RECNUM, K) ADDRESS;
716 2      (1,CNT) BYTE;
717 2      (3,4) ADDRESS DATA (10000,1000,100,10);
718 2      BUFF(5) BYTE;
719 2      RECNUM#SHL(FCB#BYTE#A#ENTENT#OFFSET),7)-FCB#BYTE#A#REC#NO;
720 3      DO I#0 TO 3;
721 4      CNT#0;
722 4      DO WHILE RECNUM#(K #J(1));
723 5      RECNUM#RECNUM - K;
724 5      CNT#CNT + 1;
725 4      END;
726 3      BUFF(I)#CNT + '0';
727 2      END;
728 2      BUFF(4)#RECNUM#'0';
729 2      IF (I #C#BYTE(8))<5 THEN
730 3      CALL MOVE(BUFF#4-I,C#ADDR(3),1);
731 3      ELSE DO;
732 4      CALL FILL(C#ADDR(8),I-5,1);
733 4      CALL MOVE(BUFF,C#ADDR(3)-I-6,5);
734 3      END;
735 2      END GET#REC#NUMBER;

736 1      WRITE#ZERO#RECORD PROCEDURE;
737 2      DO A#CTR#1 TO C#ADDR(2);
738 3      CALL WRITE#BYTE(0);
739 2      END;
740 2      END WRITE#ZERO#RECORD;

741 1      WRITE#RANDOM PROCEDURE;
742 2      CALL SET#RANDOM#POINTER;
743 2      CALL WRITE#FROM#MEMORY;
744 2      CALL INCLPTR(3);
745 2      END WRITE#RANDOM;

746 1      BACK#ONE#RECORD PROCEDURE;
747 2      CALL SET#I#0;
748 2      IF (BUFF#PTR #BUFF#PTR-(C#ADDR(2)+2)*BUFF#START THEN RETURN;
749 2      BUFF#PTR#BUFF#END-(BUFF#START - BUFF#PTR);
750 2      IF (FCB#BYTE#A#REC#NO #FCB#BYTE#A#REC#NO)-1#255 THEN
751 3      DO;
752 3      FCB#BYTE#A#ENTENT#OFFSET#FCB#BYTE#A#ENTENT#OFFSET-1;
753 3      IF OPEN(C#ADDR(0))<0 THEN
754 4      DO;
755 4      CALL PRINT#ERROR#(C#PTR);
756 4      INVALID#TRUE;
757 4      END;
758 3      FCB#BYTE#A#REC#NO#127;
759 3      END;
760 2      CALL READ#RECORD;
761 2      END BACK#ONE#RECORD;

/* * * * * * * * * * * MOVES * * * * * * * * * * */

762 1      INC#HOLD PROCEDURE;
763 2      HOLD#HOLD + 1;

```



```

762 2      CTR=CTR + 1,
763 2      END INC$HOLD,

764 1      LOAD$INC PROCEDURE,
765 3      H$BYTE(0)=0$BYTE(0),
766 2      BASE=BASE+1,
767 2      CALL INC$HOLD,
768 2      END LOAD$INC,

769 1      CHECK$EDIT PROCEDURE(CHAR),
770 2      DECLARE CHAR BYTE,
771 2      IF (CHAR=0) OR (CHAR='???') THEN CALL INC$HOLD,
772 2      ELSE IF CHAR='B' THEN
773 2      DO,
774 2      H$BYTE(0)='?',
775 3      H$BYTE(0)=H$BYTE(0),
776 3      CALL INC$HOLD,
777 3      END,
778 2      ELSE IF CHAR='A' THEN
779 2      DO,
780 3      IF NOT LETTER(0$BYTE(0)) THEN CALL PRINT$ERROR(10),
781 3      CALL LOAD$INC,
782 3      END,
783 2      ELSE IF CHAR='9' THEN
784 2      DO,
785 3      IF NOT NUMERIC(0$BYTE(0)) THEN CALL PRINT$ERROR(10),
786 3      CALL LOAD$INC,
787 3      END,
788 2      ELSE CALL LOAD$INC,
789 2      END CHECK$EDIT,
790 2      /* ***** MACHINE ACTIONS ***** */

791 1      STOP PROCEDURE,
792 2      CALL PRINT(,END OF JOB & ),
793 2      CALL BOOTER,
794 2      END STOP,

/* *****

THE PROCEDURE BELOW CONTROLS THE EXECUTION OF THE CODE.
IT DECODES EACH OP-CODE AND PERFORMS THE ACTIONS

***** */

795 1      EXECUTE PROCEDURE,
796 2      DO FOREVER,
797 3      DO CASE GET$OP$CODE,

798 4      , /* CASE ZERO NOT USED */

/* 01 ADD */

800 4      CALL ADD,

/* 02 SUB */

801 4      DO,
802 5      CALL COMPLIMENT(0),
803 5      IF SIGN(0) THEN SIGN(0)=NEGATIVE,
804 5      ELSE SIGN(0)=POSITIVE,
805 5      CALL ADD,
806 5      END,
807 5      /* 03 MUL */

808 4      DO,
809 5      DECLARE I BYTE,
810 5      CALL SET$MULT$DIV,
811 5      DECP1,DECP2=DECP1 + DECP0,
812 5      CALL ALIGN,
813 5      CALL MULTIPLY(0$R1(1+3),4),
814 5      DO INDEX=1 TO 3,
815 6      CALL MULTIPLY(R1(I+1),AND 0FH),
816 6      CALL MULTIPLY(0$R1(I),4),
817 6      END,
818 5      END,

/* 04 DIV */

819 4      CALL DIVIDE,
/* 05 NEG */

```



```

820 4          BRANCH$FLAG=NOT BRANCH$FLAG,
/* 06 STP */
821 4          CALL STOP;
/* 07 STI */
822 4          CALL STORE$IMMEDIATE,

/* 08 RND */
823 4          DO,
824 5              CALL STORE$IMMEDIATE,
825 5              CALL FILL( R2,10,0),
826 5              R2(9)=1,
827 5              CALL ADD,
828 5          END,

/* 09 RET */
829 4          DO,
830 5              IF C$ADDR(0)<0 THEN
831 5                  DO,
832 5                      A$CTR=C$ADDR(0),
833 5                      C$ADDR(0)=0,
834 5                      PROGRAM$COUNTER=A$CTR,
835 5                  END,
836 5              ELSE CALL INC$PTR(2),
837 5          END,

/* 10 CLS */
838 4          DO,
839 5              CALL SET$IFO,
840 5              IF WRITE$MARK THEN CALL WRITE$RECORD,
841 5              CALL CLOSE(C$ADDR(0)),
842 5              CALL INC$PTR(2),
843 5          END,
844 5

/* 11 SER */
845 4          DO,
846 5              IF OVERFLOW THEN PROGRAM$COUNTER = C$ADDR(0),
847 5              ELSE CALL INC$PTR(2),
848 5          END,
849 5

/* 12 BRN */
850 4          PROGRAM$COUNTER=C$ADDR(0),

/* 13 OPN */
851 4          CALL OPEN$FILE(1),

/* 14 OP1 */
852 4          CALL OPEN$FILE(2),

/* 15 OP2 */
853 4          CALL OPEN$FILE(3),

/* 16 RGT */
854 4          DO,
855 5              IF NOT SIGN(2) THEN
856 5                  BRANCH$FLAG=NOT BRANCH$FLAG,
857 5                  CALL COND$BRANCH(3),
858 5              END,

/* 17 RLT */
859 4          DO,
860 5              IF SIGN(2) THEN
861 5                  BRANCH$FLAG=NOT BRANCH$FLAG,
862 5                  CALL COND$BRANCH(3),
863 5              END,

/* 18 REQ */
864 4          DO,
865 5              IF R2$ZERO THEN
866 5                  BRANCH$FLAG=NOT BRANCH$FLAG,
867 5                  CALL COND$BRANCH(3),
868 5              END,

```



```

      * 19 INV */
869 4      CALL INCR$OR$BRANCH(INVALID),
/* 20 EOR */
870 4      CALL INCR$OR$BRANCH(END$OF$RECORD),
/* 21 ACC */
871 4      CALL ACCEPT,
/* 22 DIS */
872 4      CALL DISPLAY,
/* 23 STD */
873 4      DO,
874 5          CALL DISPLAY,
875 5          CALL STOP,
876 5      END,
/* 24 LDI */
877 4      DO,
878 5          C$ADDR(2)=CONVERT$TO$HEX(C$ADDR(0), C$BYTE(2))+1,
879 5          CALL INC$PTR(3),
880 5      END,
/* 25 DEC */
881 4      DO,
882 5          IF C$ADDR(0)<0 THEN C$ADDR(0)=C$ADDR(0)+1,
883 5          IF C$ADDR(0)=0 THEN PROGRAM$COUNTER=C$ADDR(1),
884 5          ELSE CALL INC$PTR(4),
885 5      END,
/* 26 STO */
886 4      DO,
887 5          CALL STORE$NUMERIC,
888 5          CALL INC$PTR(4),
889 5      END,
/* 27 ST1 */
890 4      DO,
891 5          CALL STORE$NUMERIC,
892 5          CALL SET$ZONE(C$ADDR(0)+C$BYTE(2)+1),
893 5      END,
/* 28 ST2 */
894 4      DO,
895 5          CALL STORE$NUMERIC,
896 5          CALL SET$ZONE(C$ADDR(0)),
897 5      END,
/* 29 ST3 */
898 4      DO,
899 5          CALL CHECK$DECIMAL,
900 5          BASE=C$ADDR(0) + C$BYTE(2) - 1,
901 5          CALL STORE$AS$CHAR(C$BYTE(2) - 1),
902 5          CALL SET$SIGN$SEP(C$ADDR(0) + C$BYTE(2) - 1),
903 5      END,
/* 30 ST4 */
904 4      DO,
905 5          CALL CHECK$DECIMAL,
906 5          BASE=C$ADDR(0) + C$BYTE(2),
907 5          CALL STORE$AS$CHAR(C$BYTE(2)-1),
908 5          CALL SET$SIGN$SEP(C$ADDR(0)),
909 5      END,
/* 31 ST5 */
910 4      DO,
911 5          CALL CHECK$DECIMAL,
912 5          R0(3)=A2(3) OR SIGN(2),
913 5          CALL MOVE( P2 - 3 - C$BYTE(2), C$ADDR(0), C$BYTE(2)),
914 5          CALL INC$PTR(4),
915 5      END,
/* 32 LDD */

```



```

918 4          CALL LOAD#NUM$LIT,
/* 33 LD1 */
919 4          CALL LOAD#NUMERIC,
/* 34 LD2 */

920 4          DO,
921 5          HOLD=C$ADDR(0),
922 5          IF CHECK$FOR$SIGN(CTR =H$BYTE(1) -C$BYTE(2)-1) THEN
923 5          DO,
924 5              CALL SET$LOAD(POSITIVE),
925 5              I=I+1,
926 5          END,
927 5          ELSE DO,
928 5              CALL SET$LOAD(NEGATIVE),
929 5              CALL LOAD$A$CHAR(CTR=ZONE),
930 5          END,
931 5          CALL LOAD#NUMBERS(C$ADDR(0), I),
932 5          END,
/* 35 LDS */

934 4          DO,
935 5          HOLD=C$ADDR(0),
936 5          IF CHECK$FOR$SIGN(H$BYTE(0)) THEN
937 5          DO,
938 5              CALL SET$LOAD(POSITIVE),
939 5              CALL LOAD#NUMBERS(C$ADDR(0), C$BYTE(2)),
940 5          END,
941 5          ELSE DO,
942 5              CALL SET$LOAD(NEGATIVE),
943 5              CALL LOAD#NUMBERS(C$ADDR(0)+1, C$BYTE(2)-1),
944 5              CALL LOAD$A$CHAR(H$BYTE(0)=ZONE),
945 5          END,
946 5          END,
/* 36 LD4 */

947 4          DO,
948 5          HOLD=C$ADDR(0),
949 5          IF H$BYTE(C$BYTE(2) - 1) = '+' THEN
950 5              CALL SET$LOAD(1),
951 5          ELSE CALL SET$LOAD(0),
952 5          CALL LOAD#NUMBERS(C$ADDR(0), C$BYTE(2) - 1),
953 5          END,
/* 37 LDS */

954 4          DO,
955 5          HOLD=C$ADDR(0),
956 5          IF(H$BYTE(0) = +) THEN CALL SET$LOAD(1),
957 5          ELSE CALL SET$LOAD(0),
958 5          CALL LOAD#NUMBERS(C$ADDR(0), C$BYTE(2)-1),
959 5          END,
/* 38 LD6 */

961 4          DO,
962 5          DECLARE I BYTE,
963 5          HOLD=C$ADDR(0),
964 5          CALL SET$LOAD(H$BYTE(1) -C$BYTE(2)-1),
965 5          BASE=BASE + 3 - I,
966 5          DO CTR = 3 TO 1,
967 5              B$BYTE(CTR)=H$BYTE(CTR),
968 5          END,
969 5          B$BYTE(CTR)=B$BYTE(CTR) AND OFSH,
970 5          CALL INC$PTR(5),
971 5          END,
/* 39 PER */

972 4          DO,
973 5          BASE=C$ADDR(1)+1,
974 5          B$ADDR(0)=C$ADDR(2),
975 5          PROGRAM$COUNTER=C$ADDR(0),
976 5          END,
/* 40 CNU */

977 4          CALL COMP#NUM$UNSIGNED,
/* 41 CNE =

```



```

978 4          CALL COMP$NUMBER$SIGN,
/* 42 CAL */
979 4          CALL COMP$ALPHA,
/* 43 RWS */
980 4          DO,
981 5          CALL BACK$ONE$RECORD,
982 5          CALL WRITE$FROM$MEMORY,
983 5          CALL INC$PTR(6),
984 5          END,
/* 44 DLS */
985 4          DO,
986 5          CALL BACK$ONE$RECORD,
987 5          CALL WRITE$ZERO$RECORD,
988 5          CALL INC$PTR(6),
989 5          END,
/* 45 RDF */
990 4          DO,
991 5          CALL READ$TO$MEMORY,
992 5          CALL INC$PTR(6),
993 5          END,
/* 46 WTF */
994 4          DO,
995 5          CALL WRITE$FROM$MEMORY,
996 5          CALL INC$PTR(6),
997 5          END,
/* 47 AVL */
998 4          CALL READ$VARIABLE,
/* 48 WVL */
999 4          CALL WRITE$VARIABLE,
/* 49 SCR */
1000 4          DO,
1001 5          SUBSCRIPT(C$BYTE(2))=
          CONVERT$TO$HEX(C$ADDR(0),C$BYTE(3)),
1002 5          CALL INC$PTR(4),
1003 5          END,
/* 50 SGT */
1004 4          CALL STRING$COMPARE(1),
/* 51 SLT */
1005 4          CALL STRING$COMPARE(0),
/* 52 SEQ */
1006 4          CALL STRING$COMPARE(2),
/* 53 MOV */
1007 4          DO,
1008 5          CALL MOVE(RES(C$ADDR(1)),RES(C$ADDR(0)),C$ADDR(2)),
1009 5          IF C$ADDR(2)<0 THEN CALL
          FILL(RES(C$ADDR(1))+C$ADDR(2),C$ADDR(3),1),
1010 5          CALL INC$PTR(8),
1011 5          END,
1012 5          END,
/* 54 RRS */
1013 4          DO,
1014 5          CALL READ$TO$MEMORY,
1015 5          CALL GET$REC$NUMBER,
1016 5          CALL INC$PTR(9),
1017 5          END,
/* 55 WRS */
1018 4          DO,
1019 5          CALL WRITE$FROM$MEMORY,
1020 5          CALL GET$REC$NUMBER,
1021 5          CALL INC$PTR(9),

```



```

1022 5          END,

/* 56: RRR */

1023 4          DO,
1024 5          CALL SET$RANDOM$POINTER,
1025 5          CALL READ$TO$MEMORY,
1026 5          CALL INC$PTR(3),
1027 5          END,

/* 57: WRR */

1028 4          CALL WRITE$RANDOM,

/* 58: RWR */

1029 4          CALL WRITE$RANDOM,

/* 59: DLR */

1030 4          DO,
1031 5          CALL SET$RANDOM$POINTER,
1032 5          CALL WRITE$ZERO$RECORD,
1033 5          CALL INC$PTR(9),
1034 5          END,

/* 60: MED */

1035 4          DO,
1036 5          CALL MOVE(C$ADDR(3), C$ADDR(8), C$ADDR(4)),
1037 5          BASE=C$ADDR(1),
1038 5          HOLD=C$ADDR(0),
1039 5          CTR=0,
1040 5          DO WHILE (CTR<C$ADDR(1))AND(CTR<C$ADDR(4)),
1041 6              CALL CHECK$EDIT(M$BYTE(3)),
1042 6          END,
1043 5          IF CTR < C$ADDR(4) THEN
1044 6              CALL FILL(HOLD, C$ADDR(4)-CTR, ' '),
1045 5          END,

/* 61: MNE */

1046 4          ; /* NULL CASE */

/* 62: GDP */

1047 4          DO,
1048 5          DECLARE OFFSET BYTE,
1049 5          OFFSET=CONVERT$TO$HEX(C$ADDR(1), C$BYTE(1)-1),
1050 5          IF OFFSET > C$BYTE(0) + 1 THEN
1051 6              DO,
1052 7                  CALL PRINT$ERROR('GDP'),
1053 7                  CALL INC$PTR(SHL(C$BYTE(0), 1) + 6),
1054 6              END,
1055 5          ELSE PROGRAM$COUNTER=C$ADDR(OFFSET + 2),
1056 5          END,

1057 4          END, /* END OF CASE STATEMENT */
1058 3          END, /* END OF DO FOREVER */
1059 2          END EXECUTE,

/* * * * * * PROGRAM EXECUTION STARTS HERE * * * * */

1060 1          BASE=CODE$START,
1061 1          PROGRAM$COUNTER=B$ADDR(3),
1062 1          CALL EXECUTE,
1063 1          END,

```

MODULE INFORMATION

```

CODE AREA SIZE      = 1B8AH      70500
VARIABLE AREA SIZE  = 08C1H      1920
MAXIMUM STACK SIZE  = 0616H      220
1542 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

ISIS-II PLM11-50 V3.1 COMPILATION OF MODULE PART2
 OBJECT MODULE PLACED IN F1 PART2.OBJ
 COMPILER INVOKED BY PLM30 F1 PART2.FLM

```

      * PAGEDLENGTH(30)
1      PART2 /* MODULE NAME */
      DO,
      /* COBOL COMPILER - PART 2 */
      /* 100H = MODULE LOAD POINT */
      /* GLOBAL DECLARATIONS AND LITERALS */
2      1 DECLARE LIT LITERALLY 'LITERALLY',
2      1 DECLARE
          HASH$TABLE$ADDR LIT '1000H' /* ADDRESS OF THE BOTTOM OF
          THE TABLES FROM PART1 */
          PASS$LEN LIT '46',
          MAX$MEMORY LIT '3200H',
          PASS$STOP LIT '3100H',
          CR LIT '13',
          LF LIT '10',
          QUOTE LIT '11H',
          POUND LIT '23H',
          TRUE LIT '1',
          FALSE LIT '0',
          FOREVER LIT 'WHILE TRUE',
          IF$FLAG BYTE INITIAL(FALSE),
4      1 DECLARE MAX$RD LITERALLY '32' /* MAX READ COUNT */
          MAX$RD LITERALLY '105' /* MAX LOOK COUNT */
          MAX$PD LITERALLY '120' /* MAX PUSH COUNT */
          MAX$SD LITERALLY '218' /* MAX STATE COUNT */
          STARTS LITERALLY '1' /* START STATE */
5      1 DECLARE READ1(*) BYTE
          DATA(0,63,5,6,9,14,16,20,22,24,26,31,32,41,42,44,45,49,53
          ,54,58,60,46,28,46,23,28,29,36,37,48,59,11,33,46,34,13,28,29,56,57
          ,48,3,1,48,23,48,57,1,56,2,30,43,27,19,33,50,52,64,16,4,26,26,39
          ,48
          ,61,55,1,15,7,12,10,51,5,9,14,16,20,22,24,26,31,41,42,44,45,49,53,
          54
          ,58,60,51,7,17,1,1,5,9,14,16,20,21,22,24,26,31,41,42,44,45,49,53
          ,54
          ,58,60,48,62,8,48,25,0,0),
6      1 DECLARE LOOK1(*) BYTE
          DATA(0,48,0,48,0,2,0,48,0,1,15,0,48,0,30,43,0,2,0,27,0,7
          ,0
          ,17,0,1,15,0,55,0,55,0,55,0,55,0,1,15,0,12,0,1,0,51,0,48,0,25,0,6,
          48
          ,0),
7      1 DECLARE APPLY1(*) BYTE
          DATA(0,0,22,0,6,0,0,77,0,0,81,0,11,66,68,74,79,0,0,1,81
          ,0
          ,3,61,0,25,0,0,0,0,57,58,59,0,0,0,0,0,0,69,0,0,0,0,0,0,5,7,8,12,
          14
          ,44,0,0,2,5,6,7,8,12,13,14,18,21,23,24,26,27,28,29,33,34,40,44,75,
          76
          ,77,80,0,9,10,17,18,49,51,54,0,5,7,8,12,14,18,44,0,52,0,20,0,0,15,
          22
          ,61,65,0,0,0,0,1,81,0,0),
8      1 DECLARE READ2(*) BYTE
          DATA(0,41,6,210,9,18,33,15,17,16,20,12,24,27,28,29,30,32
          ,33,34,37,38,31,201,85,84,201,105,107,206,85,178,194,192,193,185
          ,172
          ,210,205,207,206,209,202,129,26,191,197,86,3,25,4,189,188,21,167
          ,168
          ,166,161,162,14,5,181,201,25,85,39,169,2,11,7,164,174,184,6,9,18
          ,82
          ,15,17,18,20,23,27,28,29,30,32,33,34,37,38,164,6,13,130,131,6,9,18
          ,82,15,16,17,18,20,23,27,28,29,30,32,33,34,37,38,196,48,131,198,16
          ,8
          ,0),
9      1 DECLARE LOOK2(*) BYTE
          DATA(0,12,186,22,107,198,199,16,106,142,142,124,44,189
          ,45
          ,45,110,46,196,47,111,112,49,113,52,114,114,54,56,115,57,116,58
          ,117
          ,59,118,119,119,63,64,120,147,67,69,139,75,122,78,156,128,128,81),
10     1 DECLARE APPLY2(*) BYTE
          DATA(0,0,127,68,76,103,77,127,126,185,72,72,151,150,152
          ,177,149,122,113,104,104,136,101,101,159,162,74,160,46,65,155,155
          ,156,154,143,68,124,61,34,146,66,175,78,159,55,168,60,96,144,67,96
          ,95,175,125,150,42,90,67,90,90,215,90,90,117,179,126,98,124,89,98
          ,157,91,158,143,98,125,125,42,145,42,92,50,51,91,101,203,93,211

```



```

, 135
, 195, 195, 195, 195, 195, 195, 100, 71, 70, 208, 211, 171, 62, 99, 215, 163, 100
, 140
, 141, 101, 101, 147, 82),
11 1 DECLARE INDEX1(=) BYTE
DATA(0, 1, 115, 2, 22, 115, 115, 115, 115, 23, 25, 75, 115, 115, 115,
26
, 31, 52, 115, 35, 56, 115, 44, 115, 115, 26, 115, 115, 115, 115, 23, 42, 26, 115
, 115
, 43, 44, 23, 23, 45, 115, 47, 46, 50, 115, 51, 50, 51, 54, 23, 59, 60, 23, 61, 62, 65,
66
, 66, 66, 66, 67, 68, 69, 26, 70, 26, 73, 71, 75, 91, 92, 93, 94, 95, 96, 115, 115, 117
, 119, 73, 115, 2, 26, 1, 5, 5, 7, 5, 12, 14, 17, 19, 21, 23, 25, 26, 30, 32, 34, 36, 39,
41
, 43, 45, 47, 49, 216, 125, 123, 176, 167, 160, 204, 204, 163, 170, 170, 170, 170
, 214
, 165, 1, 2, 2, 4, 4, 6, 6, 7, 7, 9, 9, 10, 10, 10, 12, 12, 12, 12, 12, 12, 12, 12, 12,
12
, 12, 12, 12, 18, 18, 18, 18, 19, 19, 19, 19, 22, 22, 22, 25, 27, 27, 27, 28, 28, 29, 29
, 29, 30, 30, 34, 34, 35, 35, 36, 36, 37, 37, 38, 38, 39, 39, 39, 40, 42, 43, 43, 44, 44
, 45, 45, 46, 46, 46, 47, 47, 54, 55, 60, 60, 60, 68, 96, 96, 98, 98, 98, 100, 100, 100
, 101, 101, 106, 106, 107, 107, 108, 111),
12 1 DECLARE INDEX2(=) BYTE
DATA(0, 1, 1, 20, 1, 1, 1, 1, 1, 2, 1, 18, 1, 1, 1, 5, 1, 5, 1, 1, 6, 1, 1, 1
1
, 5, 1, 1, 1, 1, 2, 1, 5, 1, 1, 1, 1, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 5, 2, 1, 1, 2, 1, 1, 1
, 1
, 1, 1, 1, 1, 5, 1, 5, 18, 2, 16, 1, 1, 1, 1, 1, 19, 1, 2, 2, 1, 16, 1, 20, 5, 2, 2, 2, 1, 3, 1,
3
, 2, 2, 2, 2, 3, 2, 2, 2, 2, 3, 2, 2, 2, 2, 3, 12, 22, 36, 44, 45, 47, 49, 52, 54, 56, 57,
58
, 59, 63, 64, 5, 1, 0, 0, 1, 0, 1, 2, 2, 1, 2, 0, 0, 2, 1, 0, 2, 1, 0, 2, 1, 1, 3, 5, 2, 3, 0, 1,
2
, 2, 4, 2, 5, 4, 4, 5, 1, 1, 2, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1
, 0
, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 1
, 0),
/* END OF TABLES */
13 1 DECLARE
/* JOINT DECLARATIONS */
/* THE FOLLOWING ITEMS ARE DECLARED TOGETHER IN THIS
GROUP IN ORDER TO FACILITATE THEIR BEING PASSED FROM
THE FIRST PART OF THE COMPILER.
*/
OUTPUT$FCB (33) BYTE,
DEBUGGING BYTE,
PRINT$PROD BYTE,
PRINT$TOKEN BYTE,
LIST$INPUT BYTE,
SEQ$NUM BYTE,
NEXT$SYM ADDRESS,
POINTER ADDRESS, /* POINTS TO THE NEXT BYTE TO BE READ */
NEXT$AVAILABLE ADDRESS,
MAX$INT$MEM ADDRESS,
/* I/O BUFFERS AND GLOBALS */
IN$ADDR ADDRESS INITIAL (5CH),
INPUT$CB BASED IN$ADDR (33) BYTE,
OUTPUT$BUFF (128) BYTE,
OUTPUT$PTR ADDRESS,
OUTPUT$END ADDRESS,
OUTPUT$CHAR BASED OUTPUT$PTR BYTE,
/* MESSAGES FOR OUTPUT */
14 1 DECLARE
ERROR$NEAR$(=) BYTE DATA (' ERROR NEAR '),
END$OF$PART$2(=) BYTE DATA (' END OF COMPILATION '),
/* GLOBAL COUNTERS */
15 1 DECLARE
CTR BYTE,
A$CTR ADDRESS,
BASE ADDRESS,
B$BYTE BASED BASE BYTE,
B$ADDR BASED BASE ADDRESS,
16 1 MON1 PROCEDURE (F, A) EXTERNAL,
17 2 DECLARE F BYTE, A ADDRESS,
18 2 END MON1,
19 1 MON2 PROCEDURE (F, A) BYTE EXTERNAL,
20 2 DECLARE F BYTE, A ADDRESS,
21 2 END MON2,

```



```

22 1      BOOT PROCEDURE EXTERNAL,
23 2      END BOOT,

24 1      PRINTCHAR PROCEDURE (CHAR),
25 2      DECLARE CHAR BYTE,
26 2      CALL MON1 (2, CHAR),
27 2      END PRINTCHAR,

28 1      CRLF PROCEDURE,
29 2      CALL PRINTCHAR(CR),
30 2      CALL PRINTCHAR(LF),
31 2      END CRLF,

32 1      PRINT PROCEDURE (A),
33 2      DECLARE A ADDRESS,
34 2      CALL MON1 (9, A),
35 2      END PRINT,

36 1      PRINT$ERROR PROCEDURE (CODE),
37 2      DECLARE CODE ADDRESS,
38 2      CALL CRLF,
39 2      CALL PRINTCHAR(HIGH(CODE)),
40 2      CALL PRINTCHAR(LOW(CODE)),
41 2      END PRINT$ERROR,

42 1      FATAL$ERROR PROCEDURE(Reason),
43 2      DECLARE REASON ADDRESS,
44 2      CALL PRINT$ERROR(Reason),
45 2      CALL TIME(10),
46 2      CALL BOOT,
47 2      END FATAL$ERROR,

48 1      CLOSE PROCEDURE,
49 2      IF MON2(16, OUTPUT$FCB)=255 THEN CALL FATAL$ERROR('CL'),
51 2      END CLOSE,

52 1      MORE$INPUT PROCEDURE BYTE,
53 2      /* READS THE INPUT FILE AND RETURNS TRUE IF A RECORD
54 2      WAS READ. FALSE IMPLIES END OF FILE */
55 2      DECLARE DONT BYTE,
56 2      IF (DONT =MON2(20, INPUT$FCB))>1 THEN CALL FATAL$ERROR('EP'),
57 2      RETURN NOT(DONT),
58 2      END MORE$INPUT,

58 1      WRITE$OUTPUT PROCEDURE (LOCATION),
59 2      /* WRITES OUT A 128 BYTE BUFFER FROM LOCATION */
60 2      DECLARE LOCATION ADDRESS,
61 2      CALL MON1(26, LOCATION) /* SET DMA */
62 2      IF MON2(21, OUTPUT$FCB)<0 THEN CALL FATAL$ERROR('WR'),
63 2      CALL MON1(26, 80H), /*RESET DMA */
64 2      END WRITE$OUTPUT,

65 1      MOVE PROCEDURE(SOURCE, DESTINATION, COUNT),
66 2      /* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
67 2      DECLARE (SOURCE, DESTINATION) ADDRESS,
68 2      (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION, COUNT) BYTE,
69 2      DO WHILE (COUNT =COUNT - 1) < 255,
70 2      D$BYTE=S$BYTE,
71 2      SOURCE=SOURCE +1,
72 2      DESTINATION = DESTINATION + 1,
73 2      END,
74 2      END MOVE,

75 1      FILL PROCEDURE(ADDR, CHAR, COUNT),
76 2      /* MOVES CHAR INTO ADDR FOR COUNT BYTES */
77 2      DECLARE ADDR ADDRESS,
78 2      (CHAR, COUNT, DEST BASED ADDR) BYTE,
79 2      DO WHILE (COUNT =COUNT -1)<255,
80 2      DEST=CHAR,
81 2      ADDR=ADDR + 1,
82 2      END,
83 2      END FILL,

84 1      /* * * * * * SCANNER LITS * * * * */
85 1      DECLARE
86 2      LITERAL      LIT      29,
87 2      INPUT$STP    LIT      48,
88 2      PERIOD       LIT      11,
89 2      PPARIN       LIT      13,
90 2      LPARIN       LIT      13,
91 2      INVALID      LIT      10,

92 1      /* * * * * * SCANNER TABLES * * * * */
93 1      DECLARE TOKEN$TABLE (N) BYTE DATA
94 2      /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST RESERVED WORD
95 2      FOR EACH LENGTH OF WORD */

```



```

(0,0,3,7,13,29,41,48,56,68,83),

TABLE (**) BYTE DATA (BY, 'GO', 'IF', 'TOP', 'EOF', 'ADD', 'END', 'I=0',
'NOT', 'RUN', 'CALL', 'ELSE', 'EXIT', 'FROM', 'INTO', 'LESS', 'MOVE',
'NEXT', 'OPEN', 'PAGE', 'READ', 'SIZE', 'STOP', 'THRU', 'ZERO',
'AFTER', 'CLOSE', 'ENTER', 'EQUAL', 'ERROR', 'INPUT', 'QUOTE', 'SPACE',
'TIMES', 'UNTIL', 'USING', 'WRITE', 'ACCEPT', 'BEFORE', 'DELETE',
'DIVIDE', 'OUTPUT', 'DISPLAY', 'GREATER',
'INVALID', 'NUMERIC', 'PERFORM', 'REWRITE', 'ROUNDED', 'SECTION',
'DIVISION', 'MULTIPLY', 'SENTENCE', 'SUBTRACT', 'ADVANCING',
'DEPENDING', 'PROCEDURE', 'ALPHABETIC'),

OFFSET (1) ADDRESS INITIAL
/* NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH LENGTH */
(0,0,0,8,26,36,146,176,232,264,291),

WORD$COUNT (**) BYTE DATA
/* NUMBER OF WORDS OF EACH SIZE */
(0,0,4,6,15,12,5,8,4,3,1),

MAX$ID$LEN LIT '12',
MAX$LEN LIT '10',
ADD$END (**) BYTE DATA ('EOF'),
LOOKED BYTE INITIAL (0),
HOLD BYTE,
BUFFER$END ADDRESS INITIAL (100H),
NEXT BASED POINTER BYTE,
INBUFF LIT '80H',
CHAR BYTE INITIAL(' '),
ACCU (31) BYTE,
DISPLAY (74) BYTE INITIAL (0),
TOKEN BYTE /*RETURNED FROM SCANNER */

/* PROCEDURES USED BY THE SCANNER */

32 1 NEXT$CHAR. PROCEDURE BYTE,
33 2 IF LOOKED THEN
34 2 DO
35 3 LOOKED=FALSE,
36 3 RETURN (CHAR:=HOLD),
37 3
38 2 IF (POINTER =POINTER + 1) >= BUFFER$END THEN
39 2 DO
40 3 IF NOT MORE$INPUT THEN
41 3 DO,
42 4 BUFFER$END= MEMORY,
43 4 POINTER= ADD$END,
44 4
45 3 END,
46 3 ELSE POINTER=INBUFF,
47 3
48 2 END,
49 2 RETURN (CHAR =NEXT),
50 2 END NEXT$CHAR,

51 1 GET$CHAR. PROCEDURE,
52 2 /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS NEEDED WITHOUT
53 2 THE DIRECT RETURN OF THE CHARACTER*/
54 2 CHAR=NEXT$CHAR,
55 2 END GET$CHAR,

56 1 DISPLAY$LINE. PROCEDURE,
57 2 IF NOT LIST$INPUT THEN RETURN,
58 2 DISPLAY(DISPLAY(0) + 1) = '$',
59 2 CALL PRINT( DISPLAY(1)),
60 2 DISPLAY(0)=0,
61 2 END DISPLAY$LINE,

62 1 LOAD$DISPLAY. PROCEDURE,
63 2 IF DISPLAY(0)<72 THEN
64 2 DISPLAY(DISPLAY(0) =DISPLAY(0)+1)=CHAR,
65 2 CALL GET$CHAR,
66 2 END LOAD$DISPLAY,

67 1 PUT. PROCEDURE,
68 2 IF ACCU(0) < 30 THEN
69 2 ACCU(ACCU(0) =ACCU(0)+1)=CHAR,
70 2 CALL LOAD$DISPLAY,
71 2 END PUT,

72 1 EAT$LINE. PROCEDURE,
73 2 DO WHILE CHAR<CR,
74 3 CALL LOAD$DISPLAY,
75 3
76 2 END,
77 2 END EAT$LINE,

78 1 GET$NO$BLANK. PROCEDURE,

```



```

125 2      DECLARE (N,I) BYTE.
126 2      DO FOREVER,
127 3          IF CHAR = ' ' THEN CALL LOAD$DISPLAY,
128 3          ELSE
129 3              IF CHAR=CR THEN
130 3                  DO,
131 4                      CALL DISPLAY$LINE,
132 4                      IF SEQ$NUM THEN N=0; ELSE N=2,
133 4                      DO I = 1 TO N,
134 4                          CALL LOAD$DISPLAY,
135 4                      END,
136 4                      IF CHAR = ' ' THEN CALL EAT$LINE,
137 4                      END,
138 4                      ELSE
139 4                          IF CHAR = ' / ' THEN
140 4                              DO,
141 4                                  IF NOT DEBUGGING THEN CALL EAT$LINE,
142 4                                  ELSE
143 4                                      CALL LOAD$DISPLAY,
144 4                                      END,
145 4                                  ELSE
146 4                                      RETURN,
147 4                                  /* END OF DO FOREVER */
148 4                                  END GET$NO$BLANK.
149 2
150 1      SPACE. PROCEDURE BYTE,
151 2      RETURN (CHAR=' ') OR (CHAR=CR),
152 2      END SPACE,
153 1      LEFT$PARIN. PROCEDURE BYTE,
154 2      RETURN CHAR = '(',
155 2      END LEFT$PARIN,
156 1      RIGHT$PARIN. PROCEDURE BYTE,
157 2      RETURN CHAR = ')',
158 2      END RIGHT$PARIN,
159 1      DELIMITER. PROCEDURE BYTE,
160 2      /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR=
161 2      IF CHAR <> '.' THEN RETURN FALSE,
162 2      HOLD=NEXT$CHAR,
163 2      LOOKED=TRUE,
164 2      IF SPACE THEN
165 2          DO,
166 3              CHAR = ' ',
167 3              RETURN TRUE,
168 3          END,
169 2          CHAR=' ',
170 2          RETURN FALSE,
171 2      END DELIMITER,
172 1      END$DF$TOKEN. PROCEDURE BYTE,
173 2      RETURN SPACE OR DELIMITER OR LEFT$PARIN OR RIGHT$PARIN,
174 2      END END$DF$TOKEN,
175 1      GET$LITERAL. PROCEDURE BYTE,
176 2      CALL LOAD$DISPLAY,
177 2      DO FOREVER,
178 3          IF CHAR = QUOTE THEN
179 3              DO,
180 4                  CALL LOAD$DISPLAY,
181 4                  RETURN LITERAL,
182 4              END,
183 3              CALL PUT,
184 3          END,
185 2      END GET$LITERAL,
186 1      LOOK$UP. PROCEDURE BYTE,
187 2      DECLARE POINT ADDRESS,
188 2      HERE BASED POINT (1) BYTE, I BYTE,
189 2      MATCH. PROCEDURE BYTE,
190 3      DECLARE J BYTE,
191 3      DO J=1 TO ACCUM(0),
192 4          IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE,
193 4      END,
194 3      RETURN TRUE,
195 2      END MATCH,
196 2      POINT=OFFSET(ACCUM(0))- TABLE,
197 2      DO I=1 TO WORD$COUNT(ACCUM(0)),
198 3          IF MATCH THEN RETURN I,
199 3          POINT = POINT + ACCUM(0),
200 3      END,
201 2      RETURN FALSE,
202 2      END LOOK$UP,
203 2

```



```

204 1  RESERVED$WORD  PROCEDURE BYTE,
      /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE CONTENTS OF
      THE ACCUMULATOR IS A RESERVED WORD, OTHERWISE RETURNS ZERO */
205 2  DECLARE VALUE BYTE,
206 2  DECLARE NUMB BYTE,
207 2  IF ACCUM(0) <= MAX$LEN THEN
208 2  DO,
209 3  IF (NUMB =TOKEN$TABLE(ACCUM(0)))<>0 THEN
210 3  DO,
211 4  IF (VALUE =LOOK$UP) <> 0 THEN
212 4  NUMB=NUMB + VALUE,
213 4  ELSE NUMB=0,
214 4  END,
215 3  END,
216 2  RETURN NUMB,
217 2  END RESERVED$WORD,

218 1  GET$TOKEN  PROCEDURE BYTE,
219 2  ACCUM(0)=0,
220 2  CALL GETIND$BLANK,
221 2  IF CHAR=QUOTE THEN RETURN GET$LITERAL,
222 2  IF DELIMITER THEN
223 2  DO,
224 3  CALL PUT,
225 3  RETURN PERIOD,
226 3  END,
227 3  IF LEFT$PARIN THEN
228 3  DO,
229 4  CALL PUT,
230 4  RETURN LPARIN,
231 4  END,
232 3  IF RIGHT$PARIN THEN
233 3  DO,
234 4  CALL PUT,
235 4  RETURN RPARIN,
236 4  END,
237 3  DO FOREVER,
238 4  CALL PUT,
239 4  IF END$OF$TOKEN THEN RETURN INPUT$STR,
240 4  END, /* OF DO FOREVER */
241 2  END GET$TOKEN,

      /*  END OF SCANNER ROUTINES  */

      /*  SCANNER EXEC  */

244 1  SCANNER  PROCEDURE,
245 2  IF (TOKEN =GET$TOKEN) = INPUT$STR THEN
246 2  IF (CTR =RESERVED$WORD) <> 0 THEN TOKEN=CTR,
247 2
248 2  END SCANNER,

249 1  PRINT$ACCUM  PROCEDURE,
250 2  ACCUM(ACCUM(0)+1)=$',
251 2  CALL PRINT( ACCUM(1)),
252 2  END PRINT$ACCUM,

253 1  PRINT$NUMBER  PROCEDURE(NUMB),
254 2  DECLARE (NUMB,I,CNT,K) BYTE, J (=> BYTE DATA(100,10),
255 2  DO I=0 TO 1,
256 3  CNT=0,
257 3  DO WHILE NUMB >= (K =J(I)),
258 4  NUMB=NUMB - K,
259 4  CNT=CNT + 1,
260 4  END,
261 3  CALL PRINTCHAR('0' + CNT),
262 3  END,
263 2  CALL PRINTCHAR('0' + NUMB),
264 2  END PRINT$NUMBER,

      /*  * * *  END OF SCANNER PROCEDURES  * * *  */

      /*  * * *  SYMBOL TABLE DECLARATIONS  * * *  */

265 1  DECLARE

      CUR$SYM          ADDRESS,          /*=SYMBOL BEING ACCESSED=*/
      SYMBOL           BASED CUR$SYM (1) BYTE,
      SYMBOL$ADDR      BASED CUR$SYM (1) ADDRESS,
      NEXT$SYM$ENTRY   BASED NEXT$SYM    ADDRESS,
      HASH$MASK        LIT              '0FH',
      S$TYPE           LIT              '2',
      DISPLACEMENT     LIT              12,

```



```

OCCURS          LIT          '11',
P$LENGTH        LIT          '12',
P$L$LENGTH      LIT          '13',
LEVEL           LIT          '10',
REL$ID          LIT          '5',
LOCATION          LIT          '2',
START$NAME      LIT          '11', /*1 LESS*/
FCB$ADDR        LIT          '4',

/* * * * * * SYMBOL TYPE LITERALS * * * * */

UNRESOLVED      LIT          '255',
LABEL$TYPE      LIT          '32',
MULT$OCCURS     LIT          '128',
GROUP           LIT          '6',
NON$NUMERIC$LIT LIT          '7',
ALPHA           LIT          '8',
ALPHA$NUM       LIT          '9',
LIT$SPACE       LIT          '10',
LIT$QUOTE       LIT          '11',
LIT$ZERO        LIT          '12',
NUMERIC$LITERAL LIT          '13',
NUMERIC         LIT          '16',
COMP            LIT          '21',
A$EO            LIT          '72',
A$N$EO          LIT          '73',
NUM$EO          LIT          '80',

/* * * * * SYMBOL TABLE ROUTINES * * * * */

266 1  SET$ADDRESS PROCEDURE(ADDR);
267 2  DECLARE ADDR ADDRESS,
268 2  SYMBOL$ADDR(LOCATION)=ADDR;
269 2  END SET$ADDRESS;

270 1  GET$ADDRESS PROCEDURE ADDRESS;
271 2  RETURN SYMBOL$ADDR(LOCATION);
272 2  END GET$ADDRESS;

273 1  GET$FCB$ADDR PROCEDURE ADDRESS;
274 2  RETURN SYMBOL$ADDR(FCB$ADDR);
275 2  END GET$FCB$ADDR;

276 1  GET$TYPE PROCEDURE BYTE;
277 2  RETURN SYMBOL($TYPE);
278 2  END GET$TYPE;

279 1  SET$TYPE PROCEDURE(TYPE);
280 2  DECLARE TYPE BYTE,
281 2  SYMBOL($TYPE)=TYPE;
282 2  END SET$TYPE;

283 1  GET$LENGTH PROCEDURE ADDRESS;
284 2  RETURN SYMBOL$ADDR(PLO$LENGTH);
285 2  END GET$LENGTH;

286 1  GET$LEVEL PROCEDURE BYTE;
287 2  RETURN SHR(SYMBOL(LEVEL),4);
288 2  END GET$LEVEL;

289 1  GET$DECIMAL PROCEDURE BYTE;
290 2  RETURN SYMBOL(LEVEL) AND 0FH;
291 2  END GET$DECIMAL;

292 1  GET$P$LENGTH PROCEDURE BYTE;
293 2  RETURN SYMBOL(P$LENGTH);
294 2  END GET$P$LENGTH;

295 1  BUILD$SYMBOL PROCEDURE(LEN);
296 2  DECLARE LEN BYTE, TEMP ADDRESS,
297 2  TEMP=NEXT$SYM;
298 2  IF (NEXT$SYM = SYMBOL(LEN=LEN+DISPLACEMENT))
299 2  > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
300 2  CALL FILL(TEMP,0,LEN);
301 2  END BUILD$SYMBOL;

302 1  AND$OUT$OCCURS PROCEDURE (TYPE$IN) BYTE;
303 2  DECLARE TYPE$IN BYTE,
304 2  RETURN TYPE$IN AND 127;
305 2  END AND$OUT$OCCURS;

/* * * * * * PARSER DECLARATIONS * * * * */
306 1  DECLARE

```



```

PSTACKSIZE LIT '30', /* SIZE OF PARSE STACKS*/
VALUE (PSTACKSIZE) ADDRESS, /* TEMP VALUES */
STATESTACK (PSTACKSIZE) BYTE, /* SAVED STATES */
VALUE2 (PSTACKSIZE) ADDRESS, /* VALUE2 STACK*/
VARC (100) BYTE, /*TEMP CHAR STORE*/
IO$STACK (20) ADDRESS,
IO$PTR BYTE,
MAX$BYTE BASED MAX$INT$MEM BYTE,
SUB$INO BYTE INITIAL (0),
CONO$TYPE BYTE,
HOLD$SECTION ADDRESS,
HOLD$SEC$ADDR ADDRESS,
SECTION$FLAG BYTE INITIAL (0),
L$ADDR ADDRESS,
L$LENGTH ADDRESS,
L$TYPE BYTE,
L$DEC BYTE,
CON$LENGTH BYTE,
COMPILING BYTE INITIAL(TRUE),
SP BYTE INITIAL (255),
MP BYTE,
MPP1 BYTE,
NOLOOK BYTE INITIAL(FALSE),
(I,J,K) BYTE, /*INDICES FOR THE PARSER*/
STATE BYTE INITIAL(STARTS),

```

/* * * * * * CODE LITERALS * * * * * */

/* THE CODE LITERALS ARE BROKEN INTO GROUPS DEPENDING
ON THE TOTL LENGTH OF CODE PRODUCED FOR THAT ACTION */

/* LENGTH ONE */

```

ADD LIT '1', /* REGISTER ADDITION */
SUB LIT '2', /* REGISTER SUBTRACTION */
MUL LIT '3', /* REGISTER MULTIPLICATION */
DIV LIT '4', /* REGISTER DIVISION */
NEG LIT '5', /* NOT OPERATOR */
STP LIT '6', /* STOP PROGRAM */
STI LIT '7', /* STORE REGISTER 1 INTO REGISTER 0 */

```

/* LENGTH TWO */

```

RND LIT '8', /* ROUND CONTENTS OF REGISTER 1 */

```

/* LENGTH THREE */

```

RET LIT '9', /* RETURN */
CLS LIT '10', /* CLOSE */
SER LIT '11', /* SIZE ERROR */
BRN LIT '12', /* BRANCH */
OPN LIT '13', /* OPEN FOR INPUT */
OP1 LIT '14', /* OPEN FOR OUTPUT */
OP2 LIT '15', /* OPEN FOR I-O */
RGT LIT '16', /* REGISTER GREATER THAN */
ALT LIT '17', /* REGISTER LESS THAN */
REQ LIT '18', /* REGISTER EQUAL */
INV LIT '19', /* INVALID FILE ACTION */
EOR LIT '20', /* END OF FILE REACHED */

```

/* LENGTH FOUR */

```

ACC LIT '21', /* ACCEPT */
DIS LIT '22', /* DISPLAY */
STD LIT '23', /* STOP AND DISPLAY */
LOI LIT '24', /* LOAD COUNTER IMMEDIATE */

```

/* LENGTH FIVE */

```

DEC LIT '25', /* DECREMENT AND BRANCH IF ZERO */
STO LIT '26', /* STORE NUMERIC */
ST1 LIT '27', /* STORE SIGNED NUMERIC TRAILING */
ST2 LIT '28', /* STORE SIGNED NUMERIC LEADING */
ST3 LIT '29', /* STORE SEPARATE SIGN LEADING */
ST4 LIT '30', /* STORE SEPARATE SIGN TRAILING */
ST5 LIT '31', /* STORE COMPUTATIONAL */

```

/* LENGTH SIX */

```

LOD LIT '32', /* LOAD NUMERIC LITERAL */
LD1 LIT '33', /* LOAD NUMERIC */
LD2 LIT '34', /* LOAD SIGNED NUMERIC TRAILING */
LD3 LIT '35', /* LOAD SIGNED NUMERIC LEADING */
LD4 LIT '36', /* LOAD SEPARATE SIGN TRAILING */
LD5 LIT '37', /* LOAD SEPARATE SIGN LEADING */
LO6 LIT '38', /* LOAD COMPUTATIONAL */

```

/* LENGTH SEVEN */

```

PER LIT '39', /* PERFORM */
CMU LIT '40', /* COMPARE FOR UNSIGNED NUMERIC */
CMS LIT '41', /* COMPARE FOR SIGNED NUMERIC */
CAL LIT '42', /* COMPARE FOR ALPHABETIC */

```



```

RWS LIT '43', /* REWRITE SEQUENTIAL */
CLS LIT '44', /* DELETE SEQUENTIAL */
RDF LIT '45', /* READ SEQUENTIAL */
WTF LIT '46', /* WRITE SEQUENTIAL */
RVL LIT '47', /* READ VARIABLE LENGTH */
WVL LIT '48', /* WRITE VARIABLE LENGTH */

/* LENGTH NINE */
SCR LIT '49', /* SUBSCRIPT COMPUTATION */
SGT LIT '50', /* STRING GREATER THAN */
SLT LIT '51', /* STRING LESS THAN */
SEQ LIT '52', /* STRING EQUAL */
MOV LIT '53', /* MOVE */

/* LENGTH 10 */
RRS LIT '54', /* READ RELATIVE SEQUENTIAL */
WRS LIT '55', /* WRITE RELATIVE SEQUENTIAL */
RRR LIT '56', /* READ RELATIVE RANDOM */
WRR LIT '57', /* WRITE RELATIVE RANDOM */
RWR LIT '58', /* REWRITE RELATIVE */
DLR LIT '59', /* DELETE RELATIVE */

/* LENGTH ELEVEN */
MED LIT '60', /* MOVE EDITED */

/* LENGTH THIRTEEN */
MNE LIT '61', /* MOVE NUMERIC EDITED */

/* VARIABLE LENGTH */
GDP LIT '62', /* GQ DEPENDING ON */

/* BUILD DIRECTING ONLY */
INT LIT '63', /* INITIALIZE STORAGE */
BST LIT '64', /* BACK STUFF ADDRESS */
TER LIT '65', /* TERMINATE BUILD */
SCD LIT '66', /* SET CODE START */

/* * * * * * PARSER ROUTINES * * * * */

307 1  DIGIT  PROCEDURE (CHAR) BYTE,
308 2  DECLARE CHAR BYTE,
309 2  RETURN (CHAR<='9') AND (CHAR>='0'),
310 2  END DIGIT,

311 1  LETTER PROCEDURE BYTE,
312 2  RETURN (CHAR>='A') AND (CHAR<='Z'),
313 2  END LETTER,

314 1  INVALID$TYPE. PROCEDURE,
315 2  CALL PRINT$ERROR('IT'),
316 2  END INVALID$TYPE,

317 1  BYTE$OUT. PROCEDURE (ONE$BYTE),
318 2  DECLARE ONE$BYTE BYTE,
319 2  IF (OUTPUT$PTR = OUTPUT$PTR + 1) > OUTPUT$END THEN
320 2  DO,
321 3  CALL WRITE$OUTPUT(, OUTPUT$BUFF),
322 3  OUTPUT$PTR = OUTPUT$BUFF,
323 3  END,
324 2  OUTPUT$CHAR = ONE$BYTE,
325 2  END BYTE$OUT,

326 1  ADDR$OUT. PROCEDURE (ADDR),
327 2  DECLARE ADDR ADDRESS,
328 2  CALL BYTE$OUT(LOW(ADDR)),
329 2  CALL BYTE$OUT(HIGH(ADDR)),
330 2  END ADDR$OUT,

331 1  INC$COUNT PROCEDURE (CNT),
332 2  DECLARE CNT BYTE,
333 2  IF (NEXT$AVAILABLE = NEXT$AVAILABLE + CNT)
334 2  > MAX$INT$MEM THEN CALL FATAL$ERROR('MO'),
335 2  END INC$COUNT,

336 1  ONE$ADDR$OPP PROCEDURE (CODE, ADDR),
337 2  DECLARE CODE BYTE, ADDR ADDRESS,
338 2  CALL BYTE$OUT(CODE),
339 2  CALL ADDR$OUT(ADDR),
340 2  CALL INC$COUNT(C),
341 2  END ONE$ADDR$OPP,

342 1  NOT$IMPLIMENTED PROCEDURE,
343 2  CALL PRINT$ERROR('INI'),
344 2  END NOT$IMPLIMENTED,

```



```

345 1 MATCH PROCEDURE ADDRESS,
/* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
TABLE. IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS,
OTHERWISE THE POINTERS ARE SET FOR ENTRY*/
346 2 DECLARE POINT ADDRESS, COLLISION BASED POINT ADDRESS, (HOLD, I) BYTE,
347 2 IF VARC(0)>MAX$ID$LEN THEN VARC(0)=MAX$ID$LEN,
348 2 HOLD=0,
349 2 DO I=1 TO VARC(0),
350 2 HOLD=HOLD+VARC(I),
351 2
352 2 END,
353 2 POINT=HASH$TAB$ADDR + SHL((HOLD AND HASH$MASK), 1),
354 2 DO FOREVER,
355 2 IF COLLISION=0 THEN
356 2 DO,
357 4 CUR$SYM, COLLISION=NEXT$SYM,
358 4 CALL BUILD$SYMBOL(VARC(0)),
359 4 SYMBOL(P$LENGTH)=VARC(0),
360 4 DO I=1 TO VARC(0),
361 4 SYMBOL(START$NAME+I)=VARC(I),
362 4 END,
363 4 CALL SET$TYPE(UNRESOLVED), /* UNRESOLVED LABEL */
364 4 RETURN CUR$SYM,
365 4
366 2 ELSE
367 2 DO,
368 4 CUR$SYM=COLLISION,
369 4 IF (HOLD =GET$P$LENGTH)=VARC(0) THEN
370 4 DO,
371 4 I=1,
372 4 DO WHILE SYMBOL(START$NAME + I)= VARC(I),
373 4 IF (I =I-1)>HOLD THEN RETURN(CUR$SYM)=COLLISION,
374 4 END,
375 4 END,
376 4
377 2 POINT=COLLISION,
378 2 END,
379 2 END MATCH,

380 1 SET$VALUE, PROCEDURE(NUMB),
381 2 DECLARE NUMB ADDRESS,
382 2 VALUE(MP)=NUMB,
383 2 END SET$VALUE,

384 1 SET$VALUE2, PROCEDURE(ADDR),
385 2 DECLARE ADDR ADDRESS,
386 2 VALUE2(MP)=ADDR,
387 2 END SET$VALUE2,

388 1 SUB$CNT PROCEDURE BYTE,
389 2 IF (SUB$IND =SUB$IND + 1)>8 THEN
390 2 SUB$IND=1,
391 2 RETURN SUB$IND,
392 2 END SUB$CNT,

393 1 CODE$BYTE PROCEDURE (CODE),
394 2 DECLARE CODE BYTE,
395 2 CALL BYTE$OUT(CODE),
396 2 CALL INC$COUNT(1),
397 2 END CODE$BYTE,

398 1 CODE$ADDRESS, PROCEDURE (CODE),
399 2 DECLARE CODE ADDRESS,
400 2 CALL ADDR$OUT(CODE),
401 2 CALL INC$COUNT(2),
402 2 END CODE$ADDRESS,

403 1 INPUT$NUMERIC PROCEDURE BYTE,
404 2 DO CTR=1 TO VARC(0),
405 2 IF NOT DIGIT(VARC(CTR)) THEN RETURN FALSE,
406 2 END,
407 2 RETURN TRUE,
408 2 END INPUT$NUMERIC,

409 2

410 1 CONVERT$INTEGER PROCEDURE ADDRESS,
411 2 ACTP=0,
412 2 DO CTR=1 TO VARC(0),
413 2 IF NOT DIGIT(VARC(CTR)) THEN CALL PRINT$EPROR('NN'),
414 2 A$CTR=SHL(ACTR, 2)+SHL(ACTR, 1) + VARC(CTR) - '0',
415 2 END,
416 2 RETURN ACTP,
417 2 END CONVERT$INTEGER,

```



```

419 1      BACKSTUFF PROCEDURE (ADD1,ADD2);
420 2      DECLARE (ADD1,ADD2) ADDRESS;
421 2      CALL BYTE$OUT(EST);
422 2      CALL ADDR$OUT(ADD1);
423 2      CALL ADDR$OUT(ADD2);
424 2      END BACK$STUFF;

425 1      UNRESOLVED$BRANCH PROCEDURE;
426 2      CALL SET$VALUE(NEXT$AVAILABLE + 1);
427 2      CALL ONE$ADDR$DPP(ERN,0);
428 2      CALL SET$VALUE2(NEXT$AVAILABLE);
429 2      END UNRESOLVED$BRANCH;

430 1      BACK$COND: PROCEDURE;
431 2      CALL BACK$STUFF(VALUE(SP-1),NEXT$AVAILABLE);
432 2      END BACK$COND;

433 1      SET$BRANCH PROCEDURE;
434 2      CALL SET$VALUE(NEXT$AVAILABLE);
435 2      CALL CODE$ADDRESS(0);
436 2      END SET$BRANCH;

437 1      KEEP$VALUES PROCEDURE;
438 2      CALL SET$VALUE(VALUE(SP));
439 2      CALL SET$VALUE2(VALUE2(SP));
440 2      END KEEP$VALUES;

441 1      STD$ATTRIBUTES PROCEDURE(TYPE);
442 2      DECLARE TYPE BYTE;
443 2      CALL CODE$ADDRESS(GET$FCB$ADDR);
444 2      CALL CODE$ADDRESS(GET$ADDRESS);
445 2      CALL CODE$ADDRESS(GET$LENGTH);
446 2      IF TYPE=0 THEN RETURN;
448 2      CUR$SYM=SYMBOL$ADDP(PEL$ID);
449 2      CALL CODE$ADDRESS(GET$ADDRESS);
450 2      CALL CODE$BYTE(GET$LENGTH);
451 2      END STD$ATTRIBUTES;

452 1      READ$WRITE PROCEDURE(INDEX);
453 2      DECLARE INDEX BYTE;

454 2      IF (CTR:=GET$TYPE)=1 THEN
455 2      DO,
456 2          CALL CODE$BYTE(RDF=INDEX);
457 2          CALL STD$ATTRIBUTES(0);
458 2      END;
459 2      ELSE IF CTR=2 THEN
460 2      DO,
461 2          CALL CODE$BYTE(RRS=INDEX);
462 2          CALL STD$ATTRIBUTES(1);
463 2      END;
464 2      ELSE IF CTR=3 THEN
465 2      DO,
466 2          CALL CODE$BYTE(RPR=INDEX);
467 2          CALL STD$ATTRIBUTES(1);
468 2      END;
469 2      ELSE IF CTR=4 THEN
470 2      DO,
471 2          CALL CODE$BYTE(RVL=INDEX);
472 2          CALL STD$ATTRIBUTES(0);
473 2      END;
474 2      ELSE CALL PRINT$ERROR(CTR);
475 2      END READ$WRITE;

476 1      ARITHMETIC$TYPE PROCEDURE BYTE;
477 2      IF (CL$TYPE=AND$OUT$OCCURS(CL$TYPE))=NUMERIC$LITERAL)
478 2      AND (CL$TYPE<COMP) THEN RETURN CL$TYPE = NUMERIC$LITERAL;
479 2      CALL INVALID$TYPE;
480 2      RETURN 0;
481 2      END ARITHMETIC$TYPE;

482 1      DEL$PRINT PROCEDURE(FLAG);
483 2      DECLARE FLAG BYTE;
484 2      IF CTR:=GET$TYPE=0 THEN
485 2      DO,
486 2          IF FLAG THEN CALL CODE$BYTE(4WR);
487 2          ELSE CALL CODE$BYTE(4L);

```



```

489 0          CALL SETATTRIBUTES(1);
490 0          RETURN;
491 0      END;
492 1      IF (CTR=1) AND NOT FLAG THEN CALL CODE$BYTE(0);
493 1      ELSE IF (CTR=4) AND FLAG THEN CALL CODE$BYTE(4);
494 1      ELSE CALL INVALID$TYPE;
495 1      CALL SETATTRIBUTES(0);
496 1      END DEL$AUT;

499 1  ATTRIBUTES  PROCEDURE;
500 2      CALL CODE$ADDRESS(L$ADDR);
501 2      CALL CODE$BYTE(L$LENGTH);
502 2      CALL CODE$BYTE(L$DEC);
503 2      END ATTRIBUTES;

504 1  LOAD$LID  PROCEDURE($PTR);
505 2      DECLARE $SPTR BYTE;
506 2      IF (A$CTR = VALUE($PTR)) (= NON$NUMERIC$LIT) OR
507 2      (A$CTR = NUMERIC$LITERAL) THEN
508 2          DO;
509 2              L$ADDR=VALUE2($PTR);
510 2              L$LENGTH=CON$LENGTH;
511 2              L$TYPE=A$CTR;
512 2              RETURN;
513 2          END;
514 2      IF A$CTR=LIT$ZERO THEN
515 2          DO;
516 2              L$TYPE,L$ADDR=A$CTR;
517 2              L$LENGTH=1;
518 2              RETURN;
519 2          END;
520 2      CUR$SYM=VALUE($PTR);
521 2      L$TYPE=GET$TYPE;
522 2      L$LENGTH=GET$LENGTH;
523 2      L$DEC=GET$DECIMAL;
524 2      IF (L$ADDR = VALUE2($PTR))=0 THEN L$ADDR=GET$ADDRESS;
525 2      END LOAD$LID;

526 1  LOAD$REG  PROCEDURE(REG$NO,PTR);
527 2      DECLARE (REG$NO,PTR) BYTE;
528 2      CALL LOAD$LID(PTR);
529 2      CALL CODE$BYTE(LOC+ARITHMETIC$TYPE);
530 2      CALL ATTRIBUTES;
531 2      CALL CODE$BYTE(REG$NO);
532 2      END LOAD$REG;

533 1  STORE$REG  PROCEDURE(PTR);
534 2      DECLARE PTR BYTE;
535 2      CALL LOAD$LID(PTR);
536 2      CALL CODE$BYTE(CTO + ARITHMETIC$TYPE +1);
537 2      CALL ATTRIBUTES;
538 2      END STORE$REG;

539 1  STORE$CONSTANT  PROCEDURE ADDRESS;
540 2      IF (MAX$INT$MEM = MAX$INT$MEM - VARC(0)) < NEXT$AVAILABLE
541 2      THEN CALL FATAL$ERROR(100);
542 2      CALL BYTE$OUT(INT);
543 2      CALL ADDR$OUT(MAX$INT$MEM);
544 2      CALL ADDR$OUT(CON$LENGTH+VARC(0));
545 2      DO CTR = 1 TO CON$LENGTH;
546 2          CALL BYTE$OUT(VARC(CTR));
547 2      END;
548 2      RETURN MAX$INT$MEM;
549 2      END STORE$CONSTANT;

550 1  NUMERIC$LIT  PROCEDURE BYTE;
551 2      DECLARE CHAR BYTE;
552 2      DO CTR=1 TO VARC(0);
553 2          IF NOT( DIGIT(CHAR =VARC(CTR))
554 2          OR (CHAR='-' OR (CHAR='.' OR (CHAR='E' OR (CHAR='D'
555 2          THEN RETURN FALSE;
556 2      END;
557 2      RETURN TRUE;
558 2      END NUMERIC$LIT;

559 1  FOUND$STORE  PROCEDURE;
560 2      IF VALUE($P) < 0 THEN
561 2          DO;
562 2              CALL CODE$BYTE(AND);

```



```

562 0      CALL CODE$BYTE(L$DECODE)
563 0      END,
564 2      CALL STORE$REG(SP-1),
565 2      END ROUND$STORE,

566 1      ADD$SUB PROCEDURE (INDEX)
567 2      DECLARE INDEX BYTE,
568 2      CALL LOAD$REG(0, MP1),
569 2      IF VALUE(SP-3) < 0 THEN
570 2      DO,
571 3          CALL LOAD$REG(1, SP-3),
572 3          CALL CODE$BYTE(ADD),
573 3          CALL CODE$BYTE(STI),
574 3      END,
575 2      CALL LOAD$REG(1, SP-1),
576 2      CALL CODE$BYTE(ADD + INDEX),
577 2      CALL ROUND$STORE,
578 2      END ADD$SUB,

579 1      MULT$DIV PROCEDURE (INDEX)
580 2      DECLARE INDEX BYTE,
581 2      CALL LOAD$REG(0, MP1),
582 2      CALL LOAD$REG(1, SP-1),
583 2      CALL CODE$BYTE(MUL + INDEX),
584 2      CALL ROUND$STORE,
585 2      END MULT$DIV,

586 1      CHECK$SUBSCRIPT PROCEDURE,
587 2      CUR$SYM=VALUE(MP),
588 2      IF GET$TYPE(MULT$OCCURS THEN
589 2      DO,
590 3          CALL PRINT$ERROR('IS'),
591 3          RETURN,
592 3      END,
593 2      IF INPUT$NUMERIC THEN
594 2      DO,
595 3          CALL SET$VALUE2(GET$ADDRESS + (GET$LENGTH * CONVERT$INTEGER)),
596 3          RETURN,
597 3      END,
598 2      CUR$SYM=MATCH,
599 2      IF ((CTR:=GET$TYPE)<(NUMERIC) OR (CTR)<COMP) THEN
600 2      CALL PRINT$ERROR('TE'),
601 2      CALL ONE$ADDR$OPP(SCR, GET$ADDRESS),
602 2      CALL CODE$BYTE(SUB$CNT),
603 2      CALL CODE$BYTE(GET$LENGTH),
604 2      CALL SET$VALUE2(SUB$IND),
605 2      END CHECK$SUBSCRIPT,

606 1      LOAD$LABEL PROCEDURE,
607 2      CUR$SYM=VALUE(MP),
608 2      IF (A$CTR = GET$ADDRESS) < 0 THEN
609 2      CALL BACK$STUFF(A$CTR, VALUE2(MP)),
610 2      CALL SET$ADDRESS(VALUE2(MP)),
611 2      CALL SET$TYPE(LABEL$TYPE),
612 2      IF (A$CTR = GET$FCB$ADDR) < 0 THEN
613 2      CALL BACK$STUFF(A$CTR, NEXT$AVAILABLE),
614 2      SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE,
615 2      CALL ONE$ADDR$OPP(RET, 0),
616 2      END LOAD$LABEL,

617 1      LOAD$SEC$LABEL PROCEDURE,
618 2      A$CTR=VALUE(MP),
619 2      CALL SET$VALUE(HOLD$SECTION),
620 2      HOLD$SECTION=A$CTR,
621 2      A$CTR=VALUE2(MP),
622 2      CALL SET$VALUE2(HOLD$SEC$ADDR),
623 2      HOLD$SEC$ADDR = A$CTR,
624 2      CALL LOAD$LABEL,
625 2      END LOAD$SEC$LABEL,

626 1      LABEL$ADDR$OFFSET PROCEDURE (ADDR, HOLD, OFFSET, ADDRESS,
627 2      DECLARE ADDR ADDRESS,
628 2      DECLARE (HOLD, OFFSET, CTR) BYTE,
629 2      CUR$SYM=ADDR,
630 2      IF (CTR:=GET$TYPE)=LABEL$TYPE THEN
631 2      DO,
632 3          IF HOLD THEN RETURN GET$ADDRESS,
633 3          RETURN GET$FCB$ADDR,
634 3      END,
635 2      IF CTR<UNRESOLVED THEN CALL INVALID$TYPE,
636 2      IF HOLD THEN

```



```

639 2      DO;
640 3          A$CTR=GET$ADDRESS;
641 3          CALL SET$ADDRESS(NEXT$AVAILABLE + OFFSET);
642 3          RETURN A$CTR;
643 3      END;
644 2      A$CTR=GET$FCB$ADDR;
645 2      SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE + OFFSET;
646 2      RETURN A$CTR;
647 2      END LABEL$ADDR$OFFSET;

648 1      LABEL$ADDR: PROCEDURE (ADDR, HOLD) ADDRESS;
649 2          DECLARE ADDR ADDRESS;
650 2          HOLD BYTE;
651 2          RETURN LABEL$ADDR$OFFSET (ADDR, HOLD, 1);
652 2      END LABEL$ADDR;

653 1      CODE$FOR$DISPLAY PROCEDURE (POINT);
654 2          DECLARE POINT BYTE;
655 2          CALL LOAD$F$IO(POINT);
656 2          CALL ONE$ADDR$OPP(OIS, L$ADDR);
657 2          CALL CODE$BYTE(L$LENGTH);
658 2      END CODE$FOR$DISPLAY;

659 1      A$AN$TYPE PROCEDURE BYTE;
660 2          RETURN (L$TYPE=ALPHA) OR (L$TYPE=ALPHANUM);
661 2      END A$AN$TYPE;

662 1      NOT$INTEGER PROCEDURE BYTE;
663 2          RETURN L$DEC<0>;
664 2      END NOT$INTEGER;

665 1      NUMERIC$TYPE PROCEDURE BYTE;
666 2          RETURN (L$TYPE=NUMERIC$LITERAL) AND (L$TYPE<=COMP);
667 2      END NUMERIC$TYPE;

668 1      GEN$COMPARE PROCEDURE;
669 2          DECLARE (H$TYPE, H$DEC) BYTE;
670 2          (H$ADDR, H$LENGTH) ADDRESS;
671 2          CALL LOAD$F$ID(CMP);
672 2          L$TYPE=AND$OUT$OCCURS(L$TYPE);
673 2          IF COND$TYPE=3 THEN /* COMPARE FOR NUMERIC */
674 3              DO;
675 3                  IF A$AN$TYPE OR (L$TYPE=COMP) THEN CALL INVALID$TYPE;
676 3                  CALL SET$VALUE2(NEXT$AVAILABLE);
677 3                  IF L$TYPE=NUMERIC THEN CALL CODE$BYTE(CNU);
678 3                  ELSE CALL CODE$BYTE(CNS);
679 3                  CALL CODE$ADDRESS(L$ADDR);
680 3                  CALL CODE$ADDRESS(L$LENGTH);
681 3                  CALL SET$BRANCH;
682 3              END;
683 2          ELSE IF COND$TYPE=4 THEN
684 3              DO;
685 3                  IF NUMERIC$TYPE THEN CALL INVALID$TYPE;
686 3                  CALL SET$VALUE2(NEXT$AVAILABLE);
687 3                  CALL CODE$BYTE(CAL);
688 3                  CALL CODE$ADDRESS(L$ADDR);
689 3                  CALL CODE$ADDRESS(L$LENGTH);
690 3                  CALL SET$BRANCH;
691 3              END;
692 2          ELSE DO;
693 3              IF NUMERIC$TYPE THEN CTR=1;
694 3              ELSE CTR=0;
695 3              H$TYPE=L$TYPE;
696 3              H$DEC=L$DEC;
697 3              H$ADDR=L$ADDR;
698 3              H$LENGTH=L$LENGTH;
699 3              CALL LOAD$F$ID(SP);
700 3              IF NUMERIC$TYPE THEN CTR=CTR+1;
701 3              IF CTR=2 THEN /* NUMERIC COMPARE */
702 4                  DO;
703 4                      CALL LOAD$REG(0, NP);
704 4                      CALL SET$VALUE2(NEXT$AVAILABLE-6);
705 4                      CALL LOAD$REG(1, SP);
706 4                      CALL CODE$BYTE(CSUB);
707 4                      CALL CODE$BYTE(AGT + COND$TYPE);
708 4                      CALL SET$BRANCH;
709 4                  END;
710 3              ELSE DO;
711 4                  * ALPHA NUMERIC COMPARE *

```



```

714 4      IF (H#DEC<0) OR (H#TYPE=COMP)
              OR (L#DEC<0) OR (L#TYPE=COMP)
              OR (H#LENGTH<L#LENGTH) THEN CALL INVALID#TYPE,
716 4      CALL SET#VALUE2(NEXT#AVAILABLE),
717 4      CALL CODE#BYTE(CGT+COND#TYPE),
718 4      CALL CODE#ADDRESS(H#ADDR),
719 4      CALL CODE#ADDRESS(L#ADDR),
720 4      CALL CODE#ADDRESS(H#LENGTH),
721 4      CALL SET#BRANCH,
722 4      END,
723 3      END,
724 2      END GEN#COMPARE,

725 1      MOVE#TYPE. PROCEDURE BYTE,
726 2      DECLARE
              HOLD#TYPE BYTE,
              ALPHA#NUM#MOVE      LIT '0',
              ASN#ED#MOVE          LIT '1',
              NUMERIC#MOVE         LIT '2',
              N#ED#MOVE            LIT '3',

727 2      L#TYPE=AND#OUT#OCCURS(L#TYPE),
728 2      IF((HOLD#TYPE=AND#OUT#OCCURS(GET#TYPE))=GROUP) OR (L#TYPE=GROUP)
              THEN RETURN ALPHA#NUM#MOVE,
729 2      IF HOLD#TYPE=ALPHA THEN
              IF ASN#TYPE OR (L#TYPE=ASN#ED) OR (L#TYPE=ASN#ED)
              THEN RETURN ALPHA#NUM#MOVE,
730 2      IF HOLD#TYPE=ALPHA#NUM THEN
              DO,
731 2      IF NOT#INTEGER THEN CALL INVALID#TYPE,
              RETURN ALPHA#NUM#MOVE,
732 2      END,
733 2      IF (HOLD#TYPE=NUMERIC) AND (HOLD#TYPE<COMP) THEN
              DO,
734 2      IF (L#TYPE=ALPHA) OR (L#TYPE=COMP) THEN CALL INVALID#TYPE,
              RETURN NUMERIC#MOVE,
735 2      END,
736 2      IF HOLD#TYPE=ASN#ED THEN
              DO,
737 2      IF NOT#INTEGER THEN CALL INVALID#TYPE,
              RETURN ASN#ED#MOVE,
738 2      END,
739 2      IF HOLD#TYPE=ASN#ED THEN
              IF ASN#TYPE OR (L#TYPE=COMP) THEN RETURN ASN#ED#MOVE,
740 2      IF HOLD#TYPE=NUM#ED THEN
              IF NUMERIC#TYPE OR (L#TYPE=ALPHA#NUM) THEN
              RETURN N#ED#MOVE,
741 2      CALL INVALID#TYPE,
742 2      RETURN 0,
743 2      END MOVE#TYPE,

744 1      GEN#MOVE PROCEDURE,
745 2      DECLARE
              LENGTH1 ADDRESS,
              ADDR1 ADDRESS,
              EXTRA ADDRESS,

746 2      ADD#ADD#LEN. PROCEDURE,
747 3      CALL CODE#ADDRESS(ADDR1),
748 3      CALL CODE#ADDRESS(L#ADDR),
749 3      CALL CODE#ADDRESS(L#LENGTH),
750 3      END ADD#ADD#LEN,

751 2      CODE#FOR#EDIT. PROCEDURE,
752 3      CALL ADD#ADD#LEN,
753 3      CALL CODE#ADDRESS(GET#PCB#ADDR),
754 3      CALL CODE#ADDRESS(LENGTH1),
755 3      END CODE#FOR#EDIT,

756 2      CALL LOAD#L#ID(MPP1),
757 2      CUR#SYN=VALUE(SP),
758 2      IF (ADDR1=VALUE2(SP))=0 THEN ADDR1=GET#ADDRESS,
759 2      LENGTH1=GET#LENGTH,

760 2      DO CASE MOVE#TYPE,
              /* ALPHA NUMERIC MOVE */
761 3      DO,
762 4      IF LENGTH1>L#LENGTH THEN EXTRA=LENGTH1-L#LENGTH,
763 4      ELSE DO,
764 4      EXTRA=0,
765 4      L#LENGTH=LENGTH1,
766 4      END,
767 4      CALL CODE#BYTE(MOV),

```



```

786 4          CALL ADD$PROD$LEN,
787 4          CALL CODE$ADDRESS$EXTRA),
788 4      END,

/* ALPHA NUMERIC EDITED */

789 3      DO,
790 4          CALL CODE$BYTE$MED);
791 4          CALL CODE$FOR$EDIT,
792 4      END,

/* NUMERIC MOVE */

793 3      DO,
794 4          CALL LOAD$REG(2,MPP1),
795 4          CALL STORE$REG(SP),
796 4      END,

/* NUMERIC EDITED MOVE */

797 3      DO,
798 4          CALL CODE$BYTE$MNE),
799 4          CALL CODE$FOR$EDIT,
800 4          CALL CODE$BYTE$L$DEC),
801 4          CALL CODE$BYTE$GET$DECIMAL);
802 3      END,
803 3      END,
804 2      END GEN$MOVE,

805 1      CODE$GEN PROCEDURE(PRODUCTION);
806 2      DECLARE PRODUCTION BYTE,
807 2      IF PRINT$PROD THEN
808 2          DO,
809 3              CALL CALF,
810 3              CALL PRINTCHAR(POUND),
811 3              CALL PRINT$NUMBER(PRODUCTION);
812 3          END,

813 2      DO CASE PRODUCTION;

/* PRODUCTIONS */

/* CASE 0 NOT USED */

814 3      ,

/* 1 <P-DIV> = PROCEDURE DIVISION <USING> <PROC-BODY> */

815 3      DO,
816 4          COMPILING = FALSE;
817 4          IF SECTION$FLAG THEN CALL LOAD$SEC$LABEL,
818 4          END,

/* 2 <USING> = USING <ID-STRING> */

820 3      CALL NOT$IMPLIMENTED; /* INTER PROG COMM */

/* 3 \! <EMPTY> */

821 3      , /* NO ACTION REQUIRED */

/* 4 <ID-STRING> = <ID> */

822 3      ID$STACK(ID$PTR)=VALUE(SP);
/* 5 \! <ID-STRING> <ID> */

823 3      DO,
824 4          IF(ID$PTR = ID$PTR+1)=20 THEN
825 4              DO,
826 5                  CALL PRINT$ERROR(<ID>),
827 5                  ID$PTR=19;
828 5              END,
829 4          ID$STACK(ID$PTR)=VALUE(SP);
830 4      END,

/* 6 <PROC-BODY> = <PARAGRAPH> */

831 3      , /* NO ACTION REQUIRED */

/* 7 \! <PROC-BODY> <PARAGRAPH> */

832 3      , /* NO ACTION REQUIRED */

/* 8 <PARAGRAPH> = <ID> <SENTENCE-LIST> */

833 3      DO,

```



```

834 4      IF SECTION$FLAG=0 THEN SECTION$FLAG=2;
835 4      CALL LOAD$LABEL;
836 4      END;

/*      9          \! CIDD SECTION          */

838 3      DO;
839 4          IF SECTION$FLAG<1 THEN
840 4              DO;
841 5                  IF SECTION$FLAG=2 THEN CALL PRINT$ERROR(PF);
842 5                  SECTION$FLAG=1;
843 5                  HOLD$SECTION=VALUE(MP);
844 5                  HOLD$SEC$ADDR=VALUE2(MP);
845 5                  END;
846 5                  ELSE CALL LOAD$SEC$LABEL;
847 4              END;
848 4      END;

/*      10  <SENTENCE-LIST>  := <SENTENCED>          */

849 3      ,      /* NO ACTION REQUIRED */

/*      11          \! <SENTENCE-LIST> <SENTENCED>          */

850 3      ,      /* NO ACTION REQUIRED */

/*      12  <SENTENCED>  = <IMPERATIVE>          */

851 3      ,      /* NO ACTION REQUIRED */

/*      13          \! <CONDITIONAL>          */

852 3      ,      /* NO ACTION REQUIRED */

/*      14          \! ENTER <ID> <OPT-ID>          */

853 3      CALL NOT$IMPLIMENTED,      /* LANGUAGE CHANGE */

/*      15  <IMPERATIVE>  = ACCEPT <SUBID>          */

854 3      DO;
855 4          CALL LOAD$L$ID(SP);
856 4          CALL ONE$ADDR$OPP(ACC,L$ADDR);
857 4          CALL CODE$BYTE(L$LENGTH);
858 4      END;

/*      16          \! <ARITHMETIC>          */

859 3      ,      /* NO ACTION REQUIRED */

/*      17          \! CALL <LIT> <USING>          */

860 3      CALL NOT$IMPLIMENTED,      /* INTER PROG COMM */

/*      18          \! CLOSE <ID>          */

861 3      CALL ONE$ADDR$OPP(CLS,GET$FCB$ADDR);

/*      19          \! <FILE-ACT>          */

862 3      ,      /* NO ACTION REQUIRED */

/*      20          \! DISPLAY <LIT/ID> <OPT-LIT/ID>          */

863 3      DO;
864 4          CALL CODE$FOR$DISPLAY(MP1);
865 4          IF VALUE(SP)<0 THEN CALL CODE$FOR$DISPLAY(SP);
866 4      END;

/*      21          \! EXIT <PROGRAM-ID>          */

868 3      ,      /* NO ACTION REQUIRED */

/*      22          \! GO <ID>          */

869 3      CALL ONE$ADDR$OPP(BRN,LABEL$ADDR(VALUE(SP),1));

/*      23          \! GO <IO-STRING> DEPENDING <ID>          */

870 3      DO;
871 4          CALL CODE$BYTE(GDP);
872 4          CALL CODE$BYTE(ID$PTR);
873 4          CUR$SYM=VALUE(SP);
874 4          CALL CODE$BYTE(GET$LENGTH);
875 4          CALL CODE$ADDRESS(GET$ADDRESS);
876 4          DO CTR=0 TO ID$PTR;
877 5              CALL CODE$ADDRESS(LABEL$ADDR$OFFSET(ID$STACK(ID$PTR),1,0));
878 5          END;

```



```

879 4      END,
      /*      24      \! MOVE <L1/ID> TO <SUBID>      */
880 3      CALL GEN$MOVE,
      /*      25      \! OPEN <TYPE-ACTION> <ID>      */
881 3      CALL ONE$ADDR$OPP(OPN + VALUE(MPP1), GET$FCB$ADDR),
      /*      26      \! PERFORM <ID> <THRU> <FINISH>      */
882 3      DO,
883 4          DECLARE (ADDR2, ADDR3) ADDRESS,
884 4          IF VALUE(SP-1)=0 THEN ADDR2=LABEL$ADDR$OFFSET(VALUE(MPP1), 0, 0),
885 4          ELSE ADDR2=LABEL$ADDR$OFFSET(VALUE(SP-1), 0, 0),
886 4          IF <ADDR3 =VALUE2(SP)>=0 THEN ADDR3=NEXT$AVAILABLE + 7,
887 4          ELSE CALL BACKSTUFF(VALUE(SP), NEXT$AVAILABLE + 7),
888 4          CALL ONE$ADDR$OPP(PER, LABEL$ADDR(VALUE(MPP1), 1)),
889 4          CALL CODE$ADDRESS(ADDR2),
890 4          CALL CODE$ADDRESS(ADDR3),
891 4      END,
892 4      /*      27      \! <READ-ID>      */
893 3      CALL NOT$IMPLIMENTED, /* GRAMMAR ERROR */
      /*      28      \! STOP <TERMINATED>      */
894 3      DO,
895 4          IF VALUE(SP)=0 THEN CALL CODE$BYTE(STP),
896 4          ELSE DO,
897 5              CALL ONE$ADDR$OPP(STD, VALUE2(SP)),
898 5              CALL CODE$BYTE(CON$LENGTH),
899 5          END,
900 4      END,
      /*      29      <CONDITIONAL> = <ARITHMETIC> <SIZE-ERROR>      */
      /*      29      <CONDITIONAL> = <IMPERATIVE>      */
901 3      CALL BACK$COND,
      /*      30      \! <FILE-ACT> <INVALID> <IMPERATIVE>      */
902 3      CALL BACK$COND,
      /*      31      \! <IF-NONTERMINAL> <CONDITION> <ACTION> ELSE      */
      /*      31      <IMPERATIVE>      */
903 3      DO,
904 4          CALL BACKSTUFF(VALUE(MPP1), VALUE2(SP-2)),
905 4          CALL BACKSTUFF(VALUE(SP-2), NEXT$AVAILABLE),
906 4      END,
      /*      32      \! <READ-ID> <SPECIAL> <IMPERATIVE>      */
907 3      CALL BACK$COND,
      /*      33      <ARITHMETIC> = ADD <L/ID> <OPT-L/ID> TO <SUBID>      */
      /*      33      <ARITHMETIC> = <ROUND>      */
908 3      CALL ADD$SUB(0),
      /*      34      \! DIVIDE <L/ID> INTO <SUBID> <ROUND>      */
909 3      CALL MULT$DIV(1),
      /*      35      \! MULTIPLY <L/ID> BY <SUBID> <ROUND>      */
910 3      CALL MULT$DIV(0),
      /*      36      \! SUBTRACT <L/ID> <OPT-L/ID> FROM      */
      /*      36      <SUBID> <ROUND>      */
911 3      CALL ADD$SUB(1),
      /*      37      <FILE-ACT> = DELETE <ID>      */
912 3      CALL DEL$RWT(0),
      /*      38      \! REWRITE <ID>      */
913 3      CALL DEL$RWT(1),
      /*      39      \! WRITE <ID> <SPECIAL-ACT>      */
914 3      CALL READ$WRITE(1),

```



```

/*      40  <CONDITION>  = <LIT/ID> <NOT> <COND-TYPE>      */
917 3      DO,
918 4          IF IF$FLAG THEN
919 4              DO,
920 3                  IF$FLAG=NOT IF$FLAG,          /* RESET IF$FLAG */
921 3                  CALL CODE$BYTE<NEG>,
922 3                  END,
923 4          CALL GEN$COMPARE,
924 4          END,

/*      41  <COND-TYPE>  = NUMERIC                          */
925 3      COND$TYPE=3,

/*      42              \! ALPHABETIC                      */
926 3      COND$TYPE=4,

/*      43              \! <COMPARED> <LIT/ID>              */
927 3      CALL KEEP$VALUES,

/*      44  <NOT> : = NOT                                    */
928 3      IF NOT IF$FLAG THEN
929 3          CALL CODE$BYTE<NEG>,
930 3          ELSE IF$FLAG=NOT IF$FLAG,          /* RESET IF$FLAG */

/*      45              \! <EMPTY>                          */
931 3      , /* NO ACTION REQUIRED */

/*      46  <COMPARED> = GREATER                            */
932 3      COND$TYPE=0,

/*      47              \! LESS                             */
933 3      COND$TYPE=1,

/*      48              \! EQUAL                             */
934 3      COND$TYPE=2,
.
/*      49  <ROUND> : = ROUNDED                              */
935 3      CALL SET$VALUE<1>,

/*      50              \! <EMPTY>                          */
936 3      , /* NO ACTION REQUIRED */

/*      51  <TERMINATED> : = <LITERAL>                      */
937 3      , /* NO ACTION REQUIRED */

/*      52              \! .RUN                             */
938 3      , /* NO ACTION REQUIRED - VALUE<SP> ALREADY ZERO */
/*      53  <SPECIAL> = <INVALID>                            */
939 3      , /* NO ACTION REQUIRED */

/*      54              \! END                              */
940 3      DO,
941 4          CALL SET$VALUE<2>,
942 4          CALL CODE$BYTE<EOR>,
943 4          CALL SET$BRANCH,
944 4          END,

/*      55  <OPT-ID> = <SUBID>                                */
945 3      , /* VALUE AND VALUE2 ALREADY SET */

/*      56              \!                                  */
946 3      , /* VALUE ALREADY ZERO */

/*      57  <ACTION> = <INFERATIVE>                          */
947 3      CALL UNRESOLVED$BRANCH,

/*      58              \! NEXT SENTENCE                    */

```



```

948 3      CALL UNRESOLVED$BRANCH;
      /* 59 <THRU> = THRU <ID> */
949 3      CALL KEEP$VALUES;
      /* 60 */
950 3      , /* NO ACTION REQUIRED */
      /* 61 <FINISH> = <L/ID> TIMES */
951 3      DO,
952 4          CALL LOAD$FL$ID(NP);
953 4          CALL ONE$ADDR$OPP(LDI, L$ADDR);
954 4          CALL CODE$BYTE(LENGTH);
955 4          CALL SET$VALUE(NEXT$AVAILABLE);
956 4          CALL ONE$ADDR$OPP(DEC, 0);
957 4          CALL SET$VALUE(NEXT$AVAILABLE);
958 4          CALL CODE$ADDRESS(0), END;
      /* 62 */ \! UNTIL <CONDITION>
960 3      CALL KEEP$VALUES;
      /* 63 */ \!
961 3      , /* NO ACTION REQUIRED */
      /* 64 <INVALID> = INVALID */
962 3      DO,
963 4          CALL SET$VALUE(1);
964 4          CALL CODE$BYTE(INV);
965 4          CALL SET$BRANCH;
966 4      END;
      /* 65 <SIZE-ERROR> = SIZE ERROR */
967 3      DO,
968 4          CALL CODE$BYTE(5ER);
969 4          CALL UNRESOLVED$BRANCH;
970 4      END;
      /* 66 <SPECIAL-ACT> = <WHEN> ADVANCING <HOW-MANY> */
971 3      CALL NOT$IMPLIMENTED; /* CARRAGE CONTROL */
      /* 67 */ \!
972 3      , /* NO ACTION REQUIRED */
      /* 68 <WHEN> = BEFORE */
973 3      CALL NOT$IMPLIMENTED; /* CARRAGE CONTROL */
      /* 69 */ \! AFTER
974 3      CALL NOT$IMPLIMENTED; /* CARRAGE CONTROL */
      /* 70 <HOW-MANY> = <INTEGER> */
975 3      CALL NOT$IMPLIMENTED; /* CARRAGE CONTROL */
      /* 71 */ \! PAGE
976 3      CALL NOT$IMPLIMENTED; /* CARRAGE CONTROL */
      /* 72 <TYPE-ACTION> = INPUT */
977 3      , /* NO ACTION REQUIRED - VALUE<SP> ALREADY ZERO */
      /* 73 */ \! OUTPUT
978 3      CALL SET$VALUE(1);
      /* 74 */ \! I=0
979 3      CALL SET$VALUE(2);
      /* 75 <SUBID> = <SUBSCRIPT> */
980 3      , /* VALUE AND VALUE2 ALREADY SET */
      /* 76 */ \! <ID>

```



```

981 3      ,      /* NO ACTION REQUIRED */
      /* 77 <INTEGER> = <INPUT> */
982 3      CALL SET$VALUE(CONVERT$INTEGER),
      /* 78 <ID> = <INPUT> */
983 3      DO,
984 4          CALL SET$VALUE(MATCH);
985 4          IF GET$TYPE=UNRESOLVED THEN CALL SET$VALUE2(NEXT$AVAILABLE),
986 4      END,
      /* 79 <L/ID> = <INPUT> */
987 3
988 3      DO,
989 4          IF NUMERIC$LIT THEN
990 4          DO,
991 5              CALL SET$VALUE(NUMERIC$LITERAL),
992 5              CALL SET$VALUE2(STORE$CONSTANT),
993 5          END,
994 4          ELSE CALL SET$VALUE(MATCH),
995 4      END,
      /* 80      \! <SUBSCRIPT> */
996 3      ,      /* NO ACTION REQUIRED */
      /* 81      \! ZERO */
997 3      CALL SET$VALUE(LIT$ZERO);
      /* 82 <SUBSCRIPT> = <ID> < <INPUT> > */
998 3      CALL CHECK$SUBSCRIPT,
      /* 83 <OPT-L/ID> = <L/ID> */
999 3      ,      /* NO ACTION REQUIRED */
      /* 84      \! <EMPTY> */
1000 3      ,      /* VALUE ALREADY SET */
      /* 85 <NN-LIT> = <LIT> */
1001 3      DO,
1002 4          CALL SET$VALUE(NON$NUMERIC$LIT),
1003 4          CALL SET$VALUE2(STORE$CONSTANT);
1004 4      END,
      /* 86      \! SPACE */
1005 3      CALL SET$VALUE(LIT$SPACE),
      /* 87      \! QUOTE */
1006 3      CALL SET$VALUE(LIT$QUOTE);
      /* 88 <LITERAL> = <NN-LIT> */
1007 3      ,      /* NO ACTION REQUIRED */
      /* 89      \! <INPUT> */
1008 3      DO,
1009 4          IF NOT NUMERIC$LIT THEN CALL INVALID$TYPE,
1010 4          CALL SET$VALUE(NUMERIC$LITERAL),
1011 4          CALL SET$VALUE2(STORE$CONSTANT),
1012 4      END,
      /* 90      \! ZERO */
1013 3
1014 3      CALL SET$VALUE(LIT$ZERO);
      /* 91 <LIT/ID> = <L/ID> */
1015 3      ,      /* NO ACTION REQUIRED */
      /* 92      \! <NN-LIT> */
1016 3      ,      /* NO ACTION REQUIRED */
      /* 93 <OPT-LIT/ID> = <LIT/ID> */
1017 3      ,      /* NO ACTION REQUIRED */

```



```

/*      94      \! EMPTY)
1018 3      /* NO ACTION REQUIRED */
/*      95      <PROGRAM-ID> = <ID>
1019 3      CALL NOT$IMPLIMENTED; /* INTER PRDG COMM */
/*      96      \!
1020 3      /* NO ACTION REQUIRED */
/*      97      <READ-ID> = READ <ID>
1021 3      CALL READ$WRITE(0);
/*      98      <IF-NONTERMINAL> = IF
1022 3      IF$FLAG = TRUE; /* SET IF$FLAG */
1023 3      END; /* END OF CASE STATEMENT */
1024 2      END CODE$GEN;
1025 1      GETIN1. PROCEDURE BYTE;
1026 2      RETURN INDEX1<STATE>;
1027 2      END GETIN1;
1028 1      GETIN2. PROCEDURE BYTE;
1029 2      RETURN INDEX2<STATE>;
1030 2      END GETIN2;
1031 1      INCSP. PROCEDURE;
1032 2      VALUE<SP = SP + 1>=0; /* CLEAR THE STACK WHILE INCREMENTING */
1033 2      VALUE2<SP>=0;
1034 2      IF SP >= P$STACKSIZE THEN CALL FATAL$ERROR('SO');
1035 2      END INCSP;
1037 1      LOOKAHEAD. PROCEDURE;
1038 2      IF NOLOCK THEN
1039 2      DO;
1040 3      CALL SCANNER;
1041 3      NOLOCK=FALSE;
1042 3      IF PRINT$TOKEN THEN
1043 3      DO;
1044 4      CALL CALF;
1045 4      CALL PRINT$NUMBER<TOKEN>;
1046 4      CALL PRINT$CHAR(' ');
1047 4      CALL PRINT$ACCUM;
1048 4      END;
1049 3      END;
1050 2      END LOOKAHEAD;
1051 1      NO$CONFLICT. PROCEDURE <CSTATE> BYTE;
1052 2      DECLARE <CSTATE, I, J, K> BYTE;
1053 2      J=INDEX1<CSTATE>;
1054 2      K=J + INDEX2<CSTATE> - 1;
1055 2      DO I=J TO K;
1056 3      IF READ1<I>=TOKEN THEN RETURN TRUE;
1057 3      END;
1058 2      RETURN FALSE;
1059 2      END NO$CONFLICT;
1061 1      RECOVER. PROCEDURE BYTE;
1062 2      DECLARE TSP BYTE, RSTATE BYTE;
1063 2      DO FOREVER;
1064 3      TSP=SP;
1065 3      DO WHILE TSP < 255;
1066 4      IF NO$CONFLICT<RSTATE = STATESTACK<TSP>> THEN
1067 4      DO; /* STATE WILL READ TOKEN */
1068 5      IF SP < TSP THEN SP = TSP - 1;
1069 5      RETURN RSTATE;
1070 5      END;
1071 4      TSP = TSP - 1;
1072 4      END;
1073 3      CALL SCANNER; /* TRY ANOTHER TOKEN */
1074 3      END;
1075 2      END RECOVER;
/* * * * * PROGRAM EXECUTION STARTS HERE * * */
/* * * * * INITIALIZATION */
1077 1      TOKEN=53; /* PRIME THE SCANNER WITH -PROCEDURE- */
1078 1      CALL MOVE<PASS1$TOP-PASS1$LEN, OUTPUT$FCB, PASS1$LEN>;
/* * * * * THIS SETS
OUTPUT FILE CONTROL BLOCK
TOGGLES

```



```

      READ POINTER
      NEXT SYMBOL TABLE POINTER
    */
1079 1  OUTPUT$END=(OUTPUT$PTR = OUTPUT$BUFF-1)+128.

      /* * * * * * PARSER * * * * */

1080 1  DO WHILE COMPILING.
1081 2  IF STATE <= MAXPNO THEN /* READ STATE */
1082 3  DO.
1083 4  CALL INCS.
1084 5  STATESTACK(SP) = STATE, /* SAVE CURRENT STATE */
1085 6  CALL LOOKAHEAD,
1086 7  I=GETIN1,
1087 8  J = I + GETIN2 - 1.
1088 9  DO I=1 TO J,
1089 10 IF READ1(I) = TOKEN THEN
1090 11 DO.
      /* COPY THE ACCUMULATOR IF IT IS AN INPUT
      STRING. IF IT IS A RESERVED WORD IT DOES
      NOT NEED TO BE COPIED */
      IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
        DO K=0 TO ACCUM(0),
          VARC(K)=ACCUM(K),
        ENDO.
        STATE=READ2(I),
        NOLOOK=TRUE,
        I=J,
      ENDO.
      ELSE
        IF I=J THEN
          DO.
            CALL PRINT$ERROR('NP'),
            CALL PRINT$ERROR$NEAR$,
            CALL PRINT$ACCUM,
            IF (STATE=RECOVER)=0 THEN COMPILING=FALSE.
          ENDO.
        ENDO.
1108 2  /* END OF READ STATE */
      ELSE
1109 2  IF STATE>MAXPNO THEN /* APPLY PRODUCTION STATE */
1110 3  DO.
1111 4  MP=SP - GETIN2,
1112 5  MP1=MP + 1,
1113 6  CALL CODE$GEN(STATE - MAXPNO),
1114 7  SP=MP,
1115 8  I=GETIN1,
1116 9  J=STATESTACK(SP),
1117 10 DO WHILE (K=APPLY1(I)) <> 0 AND J<K,
1118 11 I=I + 1,
1119 12 ENDO.
1120 13 IF (K=APPLY2(I))=0 THEN COMPILING=FALSE,
1121 14 STATE=K,
1122 15 ENDO.
      ELSE
1124 2  IF STATE<=MAXLNO THEN /*LOOKAHEAD STATE*/
1125 3  DO.
1126 4  I=GETIN1,
1127 5  CALL LOOKAHEAD,
1128 6  DO WHILE (K=LOOK1(I))<>0 AND TOKEN <>K,
1129 7  I=I+1,
1130 8  ENDO.
1131 9  STATE=LOOK2(I),
1132 10 ENDO.
      ELSE
1133 2  DO. /*PUSH STATES*/
1134 3  CALL INCS.
1135 4  STATESTACK(SP)=GETIN2,
1136 5  STATE=GETIN1,
1137 6  ENDO.
1138 2  ENDO. /* OF WHILE COMPILING */
1139 1  CALL BYTE$OUT(TER),
1140 1  DO WHILE OUTPUT$PTR< OUTPUT$BUFF,
1141 2  CALL BYTE$OUT(TER),
1142 3  ENDO.
1143 1  CALL CLOSE,
1144 1  CALL CALF,
1145 1  CALL PRINT( END$OF$PART$2),
1146 1  CALL $OOT,
1147 1  ENDO.

```

MODULE INFORMATION

CODE AREA SIZE = 1000H 32510

ISIS-II PL/N-80 V2.1 COMPILATION OF MODULE DECODE
 OBJECT MODULE PLACED IN F1 DECODE.OBJ
 COMPILER INVOKED BY PLN80 F1 DECODE.FLM

```

1          $PAGELENGTH(90)
          DECODE. DO.

          /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL COMPILER
          AND CONVERTS IT INTO A READABLE OUTPUT TO FACILITATE DEBUGGING */

          /* * * 100H * * * * * LOAD POINT */

2 1  DECLARE

          LIT          LITERALLY          'LITERALLY',
          BOOT          LIT          '0',
          BOOS          LIT          '5',
          FCB          ADDRESS          INITIAL (5CH),
          FCB$BYTE      BASED          FCB (1) BYTE,
          I            BYTE,
          ADDR          ADDRESS          INITIAL (100H),
          CHAR          BASED          ADDR BYTE,
          C$ADDR        BASED ADDR      ADDRESS,
          BUFF$END      LIT          '0FFH',
          FILE$TYPE (&) BYTE          DATA ('C','I','N'),

2 1  MON1. PROCEDURE (F,A),
4 2      DECLARE F BYTE, A ADDRESS,
5 2      L GO TO L /* * * PATCH TO JMP 5 * */
6 2  END MON1;

7 1  MON2. PROCEDURE (F,A) BYTE,
8 2      DECLARE F BYTE, A ADDRESS,
9 2      L GO TO L /* * * PATCH TO " JMP 5 " * * */
10 2      RETURN 0;
11 2  END MON2;

12 1  PRINT$CHAR. PROCEDURE (CHAR);
13 2      DECLARE CHAR BYTE;
14 2      CALL MON1(2,CHAR);
15 2  END PRINT$CHAR;

16 1  CRLF. PROCEDURE,
17 2      CALL PRINT$CHAR(13),
18 2      CALL PRINT$CHAR(10);
19 2  END CRLF;

20 1  P. PROCEDURE (ADDR1),
21 2      DECLARE ADDR1 ADDRESS, C BASED ADDR1 (1) BYTE,
22 2      CALL CRLF,
23 2      DO I=0 TO 2,
24 3          CALL PRINT$CHAR(C(I)),
25 3      END;
26 2      CALL PRINT$CHAR(' ');
27 2  END P;

28 1  GET$CHAR. PROCEDURE BYTE,
29 2      IF (ADDR = ADDR + 1) & BUFF$END THEN
30 2      DO
31 3          IF MON2(20,FCB) < 0 THEN
32 3          DO
33 4              CALL PC (END),
34 4              CALL TIME(10),
35 4              L GO TO L /* * * PATCH TO " JMP 0000 " * * */
36 4          END;
37 3          ADDR=50H.
38 1      END;
39 2      RETURN CHAR;
40 2  END GET$CHAR;

41 1  D$CHAR. PROCEDURE (OUTPUT$BYTE),
42 2      DECLARE OUTPUT$BYTE BYTE,
43 2      IF OUTPUT$BYTE < 10 THEN CALL PRINT$CHAR(OUTPUT$BYTE + 10H),
44 2      ELSE CALL PRINT$CHAR(OUTPUT$BYTE + 17H),
45 2  END D$CHAR;

47 1  D. PROCEDURE (COUNT),
48 2      DECLARE (COUNT,J) ADDRESS,
49 2      DO J=1 TO COUNT

```



```

50 1          CALL D$CHAR$SHR(GET$CHAR,4)),
51 1          CALL D$CHAR$CHAR AND 0FH),
52 1          CALL PRINT$CHAR(    ),
53 1      END,
54 2  END D,

55 1  PRINT$REST  PROCEDURE,
56 2      DECLARE
          F2  LIT  '9',
          F3  LIT  '9',
          F4  LIT  '21',
          F5  LIT  '24',
          F6  LIT  '32',
          F7  LIT  '33',
          F9  LIT  '43',
          F10 LIT  '54',
          F11 LIT  '60',
          F12 LIT  '61',
          GDP LIT  '62',
          INT LIT  '63',
          BST LIT  '64',
          TER LIT  '65',
          SCD LIT  '66',

57 2      IF CHAR < F2 THEN RETURN,
58 2      IF CHAR < F3 THEN DO, CALL D(1), RETURN, END,
59 2      IF CHAR < F4 THEN DO, CALL D(2), RETURN, END,
60 2      IF CHAR < F5 THEN DO, CALL D(3), RETURN, END,
61 2      IF CHAR < F6 THEN DO, CALL D(4), RETURN, END,
62 2      IF CHAR < F7 THEN DO, CALL D(5), RETURN, END,
63 2      IF CHAR < F9 THEN DO, CALL D(6), RETURN, END,
64 2      IF CHAR < F10 THEN DO, CALL D(8), RETURN, END,
65 2      IF CHAR < F11 THEN DO, CALL D(9), RETURN, END,
66 2      IF CHAR < F12 THEN DO, CALL D(10), RETURN, END,
67 2      IF CHAR < GDP THEN DO, CALL D(12), RETURN, END,
68 2      IF CHAR=GDP THEN DO,
69 2          CALL D(1), CALL D($HL(CHAR,1)+5), RETURN, END,
70 2      IF CHAR=INT THEN DO, CALL D(3), CALL D($ADDR + 1), RETURN, END,
71 2      IF CHAR=BST THEN DO, CALL D(4), RETURN, END,
72 2      IF CHAR=TER THEN DO, CALL PC('END'),
73 2          L GO TO L, /* PATCH TO "JMP 0" == */ END,
74 2      IF CHAR=SCD THEN DO, CALL D(2), RETURN, END,
75 2      IF CHAR <> 0FFH THEN CALL PC('XXX'),
76 2  END PRINT$REST,

/* PROGRAM EXECUTION STARTS HERE */

139 1  FCB$BYTE(32), FCB$BYTE(0) = 0,
140 1  DO I=0 TO 2,
141 2      FCB$BYTE(I+9)=FILE$TYPE(I),
142 2  END,

143 1  IF MON2(15,FCB)=255 THEN DO, CALL PC('ZIII'),
144 2      L GO TO L, END,
145 2      /* *** PATCH TO "JMP BOOT" *** */

146 1  DO WHILE 1,
147 2      IF GET$CHAR <= 56 THEN DO CASE CHAR,
148 3          /* CASE 0 NOT USED */
149 3          CALL PC('ADD'),
150 3          CALL PC('SUB'),
151 3          CALL PC('MUL'),
152 3          CALL PC('DIV'),
153 3          CALL PC('NEG'),
154 3          CALL PC('STP'),
155 3          CALL PC('STI'),
156 3          CALL PC('RND'),
157 3          CALL PC('RET'),
158 3          CALL PC('CLS'),
159 3          CALL PC('SER'),
160 3          CALL PC('BRN'),
161 3          CALL PC('OPN'),
162 3          CALL PC('TOP1'),
163 3          CALL PC('TOP2'),
164 3          CALL PC('PGT'),
165 3          CALL PC('RLT'),
166 3          CALL PC('REQ'),
167 3          CALL PC('INV'),
168 3          CALL PC('EOR'),
169 3          CALL PC('ACC'),
170 3          CALL PC('DIS'),
171 3          CALL PC('STD'),
172 3          CALL PC('LDI'),
173 3          CALL PC('DEC'),
174 3          CALL PC('STO'),

```



```

178 3          CALL PC('ST1');
179 4          CALL PC('ST2');
180 5          CALL PC('ST3');
181 6          CALL PC('ST4');
182 7          CALL PC('ST5');
183 8          CALL PC('L00');
184 9          CALL PC('LD1');
185 10         CALL PC('LD2');
186 11         CALL PC('LD3');
187 12         CALL PC('LD4');
188 13         CALL PC('LD4');
189 14         CALL PC('LD6');
190 15         CALL PC('PER');
191 16         CALL PC('CHU');
192 17         CALL PC('CHS');
193 18         CALL PC('CAL');
194 19         CALL PC('RWS');
195 20         CALL PC('OLS');
196 21         CALL PC('ROF');
197 22         CALL PC('WTF');
198 23         CALL PC('PVL');
199 24         CALL PC('WVL');
200 25         CALL PC('SCR');
201 26         CALL PC('SGT');
202 27         CALL PC('SLT');
203 28         CALL PC('SEQ');
204 29         CALL PC('MOV');
205 30         CALL PC('RPS');
206 31         CALL PC('WRS');
207 32         CALL PC('RRR');
208 33         CALL PC('WRR');
209 34         CALL PC('RWR');
210 35         CALL PC('DLR');
211 36         CALL PC('MED');
212 37         CALL PC('MNE');
213 38         CALL PC('GDP');
214 39         CALL PC('INT');
215 40         CALL PC('BST');
216 41         CALL PC('TER');
217 42         CALL PC('SCD');
218 43         END. /* OF CASE STATEMENT */
219 44         CALL PRINT$REST;
220 45         END. /* END OF DO WHILE */
221 46         END;

```

MODULE INFORMATION

```

CODE AREA SIZE      = 0671H    16490
VARIABLE AREA SIZE = 0013H     190
MAXIMUM STACK SIZE = 000EH     140
215 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-88 COMPILATION

LIST OF REFERENCES

1. Craig, Allen S. MICRO-COBOL An Implementation of Navy Standard Hypo-Cobol for a Micro-processor based Computer System.
2. Aho, A. V. and S. C. Johnson, LR Parsing, Computing Surveys, Vol. 6 No. 2, June 1974.
3. Bauer, F. L. and J. Eickel, editors, Compiler Construction - An Advanced Course, Lecture notes in Computer Science, Springer-Verlag, New York 1976.
4. Digital Research, An Introduction to CP/M Features and Facilities, 1976.
5. Digital Research, CP/M Interface Guide, 1976.
6. Eubanks, Gordon E. Jr. A Microprocessor Implementation of Extended Basic, Masters Thesis, Naval Postgraduate School, December 1976.
7. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.
8. Intel Corporation, 8080 Simulator Software Package, 1974.
9. Knuth, Donald E. On the Translation of Languages from Left to Right, Information and Control Vol. 8, No. 6, 1965.
10. Software Development Division, ADPE Selection Office, Department of the Navy, HYPO-COBOL, April 1975.
11. University of Toronto, Computer Systems Research Group Technical Report CSRG-2, "An Efficient LALR Parser Generator," by W. R. Lalonge, April 1971.
12. Digital Research, Symbolic Instruction Debugger User's Guide, 1978.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	3
4. Assoc. Professor G. A. Kildall, Code 52Kd Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5. Lt. M. S. Moranville, Code 52Ms Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6. ADPE Selection Office Department of the Navy Washington, D. C. 20376	1
7. P.R. Mylet 8005 Kidd St. Alexandria, Va. 22309	1

177913

Thesis

M998

Mylet

c.1

MICRO-COBOL: a subset
of Navy standard HYPO-
COBOL for micro-com-
puters.

30 APR 79

25936

Thesis

M998

Mylet

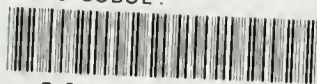
c.1

MICRO-COBOL: a subset
of Navy standard HYPO-
COBOL for micro-com-
puters.

177913

thesM998

MICRO-COBOL :



3 2768 001 92609 0

DUDLEY KNOX LIBRARY C.1